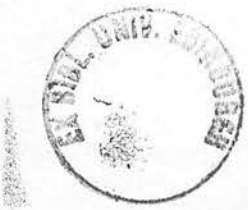
Aspects of the Constructive Omega Rule within Automated Deduction

Siani Baker

Ph.D.

University of Edinburgh

1992



Abstract

In general, cut elimination holds for arithmetical systems with the ω-rule, but not for systems with ordinary induction. Hence in the latter, there is the problem of generalisation, since arbitrary formulae can be cut in. This makes automatic theorem-proving very difficult. An important technique for investigating derivability in formal systems of arithmetic has been to embed such systems into semi-formal systems with the ω-rule. This thesis describes the implementation of such a system. Moreover, an important application is presented in the form of a new method of generalisation by means of "guiding proofs" in the stronger system, which sometimes succeeds in producing proofs in the original system when other methods fail.

I declare that this thesis has been composed by myself and that the work described in it is my own:

Siani Lynne Baker

Acknowledgements

First and foremost I would like to thank my supervisors Dr Alan Smaill and Dr Andrew Ireland for their advice, encouragement and support throughout this research project. I would also like to express my gratitude to members of the Mathematical Reasoning Group for providing such a good working and research environment, and in particular to Colin Phillips and Sue Sentance for helpful feedback on my thesis drafts. In addition I gratefully acknowledge the support of a Science and Engineering Research Council studentship.

This thesis is dedicated to my parents, for their love and encouragement over the years.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"Whatsoever we imagine is finite. Therefore, there is no idea, or conception of any thing we call infinite. No man can have in his mind an image of infinite magnitude; nor conceive infinite swiftness, infinite time, infinite force, or infinite power. When we say anything is infinite, we signify only that we are not able to conceive the ends and bounds of the thing named; having no conception of the thing, but of our own inability."*
>
> *Thomas Hobbes*

Normally, proofs considered in theorem-proving are finite; however, there is a reasonable notion of infinite proof involving the ω-rule, which infers the universal application of a proposition from an infinite number of individual cases of that proposition. It is the aim of this thesis to exploit this notion within the domain of automated theorem-proving. Necessarily, implementational problems as to the representation of such an infinite process on a computer must be overcome.

Recursively defined structures are used both in mathematics and computing, and proofs involving such structures show a certain regularity. This regularity enables proof by induction: that is a method of proving a proposition $P(n)$, containing an arbitrary positive integer n, by showing first that the proposition is true for $n = 1$ (or 0) and secondly that for any value $n = k$, $P(k)$ implies $P(k + 1)$.

If induction is used, generalisation is often needed also. Generalisation puts a proposition into a more general form, of which the original becomes a particular case. However, generalisation is a major theorem-proving problem which has not

yet been satisfactorily solved because there are no obvious heuristics which work in all cases. One possible solution to this problem is to use a stronger inference rule in place of induction. If the ω-rule is used, the problems associated with generalisation may be avoided. However, the ω-rule involves the use of infinite proofs, and therefore poses a problem as far as implementation is concerned.

With the goal of automatic derivation of proofs within some formalisation of arithmetic in mind, an (implementable) representation for an arithmetical system including the ω-rule is proposed and the benefits of such a theorem-proving system investigated. In addition, a new method of generalisation by means of "guiding proofs" in this system is proposed, which may succeed in producing proofs in Peano Arithmetic when other methods fail (in particular, by providing a suitable generalisation).

This chapter briefly discusses motivation and goals for the thesis and presents a short historical introduction to automated theorem proving, followed by discussion of the importance of automated theorem proving and generalisation. Finally, an outline of the structure of the thesis is provided.

1.1 Aims

The ω-rule allows inference of $\forall x P(x)$ from the infinite sequence of propositions $P(0), P(s(0)), P(s(s(0)))$, etc. The goals of the thesis are to implement the ω-rule and demonstrate the practical use in a theorem-proving context of the resulting system. This involves the presentation of an implementable version of the ω-rule, together with a formal description and investigation of properties of the corresponding arithmetical system. The implementational system will be seen to be useful both as a proof environment and as a guide to generalisation in the more usual formalisation of arithmetic.

1.2 Automated Theorem-Proving

This section provides a brief historical introduction to and discusses the importance of automated theorem proving.

Logic lies at the heart of artificial intelligence, since it is central to reasoning, and consists of much that is algorithmic in nature. As is revealed from the discussion in Section 2.1, whatever philosophical standpoint is taken, formal mathematical proofs still operate in terms of their own symbols, and so form an especially suitable domain for computation. As a result, theorem-proving has been a successful subject of early artificial intelligence research and implementation. Once computers became available, initial inefficient symbol-crunching approaches were improved by normalisation procedures (Gilmore, 1970), and by further refinement of standard procedures in the form of resolution (Robinson, 1965). In addition to the development of such uniform procedures, systems like PRESS (Bundy & Welham, 1981) and the Boyer-Moore theorem-prover NQTHM (Boyer & Moore, 1988) were developed with the aim of guiding proofs using various heuristics. Further developments in automated theorem proving, including advances in such high-level tactical reasoning (for example by HOL (Gordon, 1988), LCF (Gordon *et al*, 1979) and CLAM (Bundy *et al*, 1990)) are considered in Section 3.2.

Since actual human reasoning may be rather naïve, the view has been put forward, for example by Marvin Minsky and Roger Schank, that artificial intelligence should not be concerned with logic, and that automatic theorem-provers are not intelligent (Minsky, 1963). However, this is a matter of opinion: Hayes counteracts that logic is "the most successful precise language ever developed to express human thought and inference" (Hayes, 1977).

In any case, there is strong evidence for the usefulness of automated theorem-proving, for it is extremely important both from a practical and a theoretical point of view. On the practical side, both program synthesis and program verification (Constable *et al*, 1985), and hardware synthesis and hardware verification (Gordon, 1988; Basin, 1991) have arisen. In addition, theorem-provers provide

tools for mathematicians (and scientists generally), both as automatic theorem provers or proof environments. One application of the latter is for tutoring systems, which help students learn various aspects of mathematics.[1]

Theorem-proving has applications in problem solving, generally; many types of problem may be encoded by mathematics — for example, the STRIPS system reasons about change by theorem-proving methods (Fikes *et al*, 1971). The methods used are not necessarily those used by humans: there is a distinction between so-called weak artificial intelligence, which tries to simulate intelligent behaviour, without insisting that the process is analogous to an actual psychological process of some existent being, as opposed to strong artificial intelligence. The work in this thesis falls between the two camps: in particular, the (computable) method for calculating generalisations suggested in Chapter 8 would not be one normally used by a human; but on the other hand it exploits general patterns in the high-level structure of the proof in a manner which may be explained in terms of human reasoning (as opposed to some specialised technical process occurring deep within the code).

On the theoretical side, automatic theorem-provers may simulate mathematicians. There are various alternative approaches within mathematical theorem-proving, one of which is to create an 'artificial mathematician', in other words to recreate either the process or just the end result of actually doing mathematics. This involves detection of general patterns of proof, and provision of maximum possible automation of proofs. However, many implementations have chosen a less ambitious aim of producing an interactive system in order to suggest ideas to the user, to allow description and manipulation of proofs and also to automate trivial steps which would be tedious to do by hand. As mentioned above, other systems are designed with the tutoring of logic in mind; these may analyse mistakes, suggest analogies and help the user to learn various methods of proof. My work falls into the first category; the goal is to enable automation of proofs, rather than the pro-

[1]For example, (Twidale, 1989). A review of various tutoring systems is described in (Goldson *et al*, 1992).

vision of a proof environment or a tutoring system. A selection of related systems will be considered in Section 3.2.

Moreover, there are other benefits which are obtained from attempts at theorem-proving in the form of new programming languages such as Prolog (and indeed the whole field of logic programming; see for example (Hogger, 1984)). The advantages of having a formal basis for program development, compared with the approach of testing programs (which cannot be relied upon to establish either the correctness of a correct program or the incorrectness of an incorrect program) is summed up in a quotation attributed to Edsger Dijkstra (Buxton & Randell, 1969):

> "Program testing can be used to show the presence of bugs, but never to show their absence."

The acceptance of this argument entails that it is necessary to address the problems of actually carrying out formal reasoning and thus the need for automated reasoning systems.

Next, an important (and problematic) notion within theorem-proving, which is central to the issues described in this thesis, is introduced, namely that of generalisation.

1.3 Generalisation

Generalisation may be defined as a proof step which allows the postulation of a new theorem as a substitute for the current goal, from which the latter follows easily. It is important that the original should follow fairly directly, in order to ensure that the generalisation would be closely related to the original expression rather than just being a random lemma. As shall be further discussed in Section 2.7, generalisation is a powerful tool with a variety of rôles. These include defining new concepts by extension or the recombination of existing ones (which has a parallel with abstraction); turning proofs for a specific example into ones valid for a number of examples; allowing proofs (for in some systems, namely for which cut

elimination does not hold, certain theorems are only provable using generalisation); and producing clearer proofs (by the use of lemmata).

From an abstract point of view, there seem to be only a few ways of generalisation. One obvious method is the generalisation of a formula by replacing a constant with a variable. In addition, predicates and classes tend to come in families with members which may be more or less general (eg. feline and carnivore have a relationship from the more specific to the more general). This is known as ISA-hierarchy generalisation (see for example (Fahlman, 1979)).[2] Furthermore, there is numerical generalisation, such as from $x = 3$ to $x \in [2, 5]$. Other forms of generalisation include the addition of disjuncts to a condition, and universalisation, which is the process of converting lists of specific formulae to quantified combinations: for example, one might wish to generalise $P(x_1), Q(x_1), P(x_2), Q(x_2), \ldots, P(x_n), Q(x_n)$ etc. to $\forall x\ P(x) \wedge Q(x)$. An alternative approach is of considering generalisation as the inversion of sound inference rules; this will be discussed further in Section 3.1.

Van der Waerden's account of how the proof of Baudet's conjecture was found (der Waerden, 1971) illustrates how generalisation lies at the heart of mathematical discovery, and how, surprisingly, generalised theorems may be easier to prove than the original goal. This is because the induction hypothesis is also made stronger when the goal is strengthened by generalisation. Generalisation may enable the completion of the proof, but one of the inherent dangers is that a method to construct generalised conjectures may suggest non-theorems.

Although generalisation is an important problem in theorem-proving, it has by no means been solved. It is important and still being investigated for reasons which have to do with cut elimination and the lack of heuristics for providing cut formulae respectively, for in sequent calculus the generalisation step is achieved by use of the cut rule. A cut elimination theorem for a system states that every proof in that system may be replaced by one which does not involve use of the cut

---

[2]Note that generalisation may take place from individuals to classes of which they are members, or alternatively from subclasses to classes.

rule[3]. Uniform proof search methods can be used for logical systems, in sequent calculus form, where the cut rule is not used. In general, cut elimination holds for arithmetical systems with the ω-rule, but not for systems with ordinary induction. Hence in the latter, there is the problem of generalisation, since arbitrary formulae can be cut in. This makes automatic theorem-proving very difficult, especially as there is no easy or fail-safe method of generating the required cut formula. An important technique for investigating derivability in formal systems of arithmetic has been to embed such systems into semi-formal systems with the ω-rule. This thesis presents a new approach to the problems of generalisation by means of "guiding proofs" in the stronger system, which sometimes succeeds in producing proofs in the original system when other methods fail. This method has been partially automated and results in the suggestion of an appropriate cut formula.

Further discussion of generalisation is given in Section 3.1 (when various different types of generalisation are illustrated by means of examples) and Section 11.3 (in which there is a comparison of various generalisation methods).

1.4 Outline of Thesis

The structure of the thesis is as follows.

- Chapter 2 raises issues related to the notion of infinite proofs and provides motivation for the work in the thesis. After examination of the philosophical idea of infinite proofs in Section 2.1, in the context of discussing possible constructive restrictions on the ω-rule, a historical introduction to the ω-rule is given and the importance of cut elimination is highlighted. In Section 2.4 Gödel encoding is mentioned as a prelude to discussion in Chapter 4. Next, in Section 2.5 various ω-rules are introduced and compared, followed by a brief discussion of another method of providing infinite proofs, namely by

[3]See Section 2.3.

infinitary logics. Finally, in Section 2.7 various motivations for consideration of arithmetical systems with the ω-rule are given.

- Chapter 3 first discusses an analogous method of generalisation for natural deduction systems and classifies various different types of generalisation, and then describes inductive theorem proving systems and presents the notions of inductive learning, explanation-based generalisation and primitive recursion over abstract datatypes, all of which are related to and provide background information for the work described in later chapters.

- Chapter 4 presents a description of first order arithmetic with the ω-rule by extending a given system of Peano Arithmetic and discusses both the need for and possible methods of restricting such a rule. In addition, the chosen approach for restriction of the ω-rule is detailed. In order to carry out this approach without using encoding, a new notion of effectiveness must be defined.

- Chapter 5 defines such a notion over arbitrary datatypes, and shows that it is the effective prooftrees of a described arithmetical representation using the ω-rule which provide the required restriction.

- Chapter 6 presents an implemented version of such a system, which involves the application of rewrite rules, and relates this to the theoretical system described.

- Chapter 7 discusses the claim that various propositions not provable in the system of Peano Arithmetic without the cut rule are provable using the restricted ω-rule, and discusses the strength of the extended system.

- The next two chapters deal with an application of this work in the field of generalisation. Chapter 8 presents a new method of generalisation by means of which proofs in the extended system may automatically guide proofs in Peano Arithmetic. The amenability of extension of this method to other domains is shown in Section 8.2. The methods of using explanation-based

generalisation and linearisation on the individual subproofs of the ω-rule, such that a cut formula is suggested upon which induction may be performed, are expounded in Sections 8.1.4 and 8.4. Chapter 9 extends this generalisation method to other examples which are less clear, in a more speculative manner.

- Chapter 10 describes the implementation of my system (including generation of individual proofs, forming general proofs, showing correctness of the general proof, and the application of rewrite rules), a system in which the general proofs may be displayed and investigated, and also the implementation of the generalisation method (namely the provision of a cut formula via explanation-based generalisation methods) for arithmetic.

- Chapter 11 presents related work such as reflection, proof transformation and other generalisation approaches, and then moves on to give a comparison of the generalisation method suggested with these other approaches.

- Finally, in Chapter 12, suggestions for further work are provided and an assessment of the contributions of the thesis is made.

- The appendices relate to: a description of sequent calculus (Appendix A); the provision of useful commands in the Seldon system (Appendix B); a summary of various relevant proofs with respect to three of the main list generalisation examples (Appendix C); a transcript of the system NQTHM, which relates to an example considered in Subsection 11.4.1 (Appendix D); details of the implemented system (Appendix E), including transcripts illustrating the automatic generation of proofs and provision (including checking for correctness) of the general proof (Appendix E.4) plus the automatic provision of cut formulae via the explanation-based generalisation method for simple arithmetical examples (Appendix E.8.3).

Chapter 2

Infinite Proofs

> *"This infinite is potential, never actual: the number of parts that can be taken always surpasses any assigned number. But this number is not separable from the process of bisection, and its infinity is not a permanent actuality but consists in a process of coming to be, like time and the number of time."*
>
> *Aristotle*

This chapter provides the context and motivation for the topics which shall be raised in further chapters. In Section 2.1 the nature of mathematical reasoning is examined in order to clarify the notion of what is meant by an infinite proof. In so doing the nature of the ω-rule is clarified, since it generates infinite proof obligations. Section 2.2 presents a description of the historical development of the ω-rule. In Section 2.3 background issues, which involve the importance of the cut rule within automated deduction, are presented, in order to facilitate comprehension of later chapters and highlight the importance of cut elimination. The cut-elimination theorem is an important proof-theoretic result, which is especially suitable for automation. Such an implementation has been carried out for the case of predicate calculus, and is discussed in Subsection 2.3.5. Gödel encoding is discussed in Section 2.4 as a prelude to discussion in Chapter 4 of the usual approach to placing restrictions on the ω-rule. Section 2.5 introduces and compares a variety of ω-rules, and in particular the constructive ω-rule, and assesses the rôle of these rules within deduction. Another method of provision of infinite proofs, namely infinitary logic, is considered in Section 2.6. Motivation for the work in

the thesis is given in Section 2.7, and in particular Subsection 2.7.1 outlines the use of the constructive ω-rule within automated reasoning, as an introduction to further details provided in the following chapters. Finally, conclusions are given.

2.1 Notions of Infinite Proof

Since the ω-rule has infinitely many hypotheses, the nature of the resulting (infinite) proof should be considered. In Chapters 4 and 5 the extension of the arithmetical systems HA or PA (presented in Section 4.1) with a restricted form of the ω-rule will be considered as a suitable system for implementation. In this section, in order to determine how this may be a valid extension, the nature of canonical proofs within these systems will be explored. Moreover, various notions of proof and of infinite structures, and hence of infinite proofs, are discussed with reference to alternative viewpoints within the foundations of mathematics.

2.1.1 Foundational Theories Regarding Proofs

After a brief exposition of the nature of logic and a short historical introduction, various foundational views regarding proofs will be presented as a prelude to the discussion of infinite proofs, and those involving the ω-rule in particular, in the next subsection.

Nature of Logic

Logic may be traced back to Aristotle, who systematised immediate inference and syllogisms, and remained a primary reference for the subject until the last century. Since then, there have been major developments in a variety of fields including the development of the predicate calculus (of which syllogisms form a minor part), modal logics, logics of relations, deontic logics, and the formalisation of many other systems. Logic is concerned both with the definition of systems and the study of

ideas related to valid reasoning, which gives rise to the subfields of formal and philosophical logic respectively.

Formal logic is primarily concerned with axiomatisation of systems. An axiom system is a system in which certain expressions are derived in accordance with a given set of inference rules from an initial set of expressions called axioms, which are taken as given. It is necessary to specify which symbols may be used within the system and what combinations of these are to count as expressions which either may be axioms, or derived from axioms. These expressions are called formulae, and those which may be derived from the axioms are called theorems. The formation rules are analogous to grammatical rules, and the formulae to meaningful sentences. The inference rules define which formulae may be derived from others, and so determine what the theorems of the system will be, with relation to the axioms. There may be an infinite number of axioms. Important (and desirable) properties of such a system include consistency, soundness and completeness.[1] Systems complex enough to axiomatise mathematics turn out to have these properties only to a limited extent. Gödel's first and second incompleteness theorems state that any consistent formal system adequate to axiomatise arithmetic contains at least one logical formula which is neither provable nor refutable in the system, even though we may be able to see that it is true, and secondly that a system's consistency cannot be proved within the system itself (Gödel, 1931). In addition, the logical paradoxes discussed in the following section indicate difficulties in the construction of systems representing arithmetic. Interest has therefore been shown in the consideration of various systems, each with different properties.

Moreover, since the application of inference rules is (normally) computable, logic has provided a suitable domain for the use of computers; other motivations for the automation of logic have been discussed in Section 1.2.

---

[1] See Section 2.3 for further details.

The Paradoxes

Towards the end of the nineteenth century, both the theory of real numbers (defined by Dedekind as a cut of the set of rational numbers) and the theory of rational numbers and natural numbers were based on set theory, which had been introduced some half-century earlier. The axiomatisation of logic, as described in the above subsection, led to axiomatic approaches by Frege and Peano to show that arithmetic is part of logic and to formalise arithmetic respectively. But at the beginning of this century it was noted that some of the most useful notions used in set theory give rise to paradoxes. The most famous of these was Russell's paradox[2]; others include the Burali-Forti paradox in the theory of transfinite ordinals, and the Richard paradox dealing with the notion of finite definability (Kleene, 1962, P35–39). These paradoxes of set theory are analogous in form to the ancient (semantic) "liar" paradox (one formulation of which is "this statement is false"). The paradoxes showed that the concept of class or set as it was then being used had not been sufficiently clarified, since it is not possible to suppose in the face of them that a class does correspond to any statable condition on classes which may be given. Thus Frege was led to remark that the whole foundation of mathematics had been undermined. Much of the foundational research of the early twentieth-century was directed at problems posed (or thought to be posed) by the paradoxes, and three new general viewpoints were seen to emerge: Hilbert's approach, intuitionism and logicism. These foundational approaches will be discussed, but first platonism and formalism, the existing schools of thought which fell under criticism, will be considered.

Platonism

Various philosophical stances can be distinguished with regard to the nature of mathematics. Platonism is the dominant attitude in the practice of modern mathematics; there is a direct connection between platonism and the law of excluded

---

[2] "Is the set of all sets that are not members of themselves a member of itself?"

middle[3], and the classical version of arithmetic is an example of a system which includes this law. Platonists (such as Gödel) believe that numbers, sets, etc. are abstract entities, that mathematical objects have an external existence independent of our thought and that the facts concerning them do not depend in any way on the possibilities of verification.

Formalism

Formalists (as do intuitionists) deny the Platonic view of independent existence of mathematical truth. One result of the axiomatisation of a theory, as described earlier in this subsection, is that the meaning of the primitive terms become irrelevant to its deductions. Formalists, such as Bourbaki and Curry, carried this abstraction from meaning to its limit, and held that mathematics consists of formal systems whose elements are meaningless symbols, to be operated on by fixed rules. Much of the language is replaced by an artificial syntax, and what remains is a specification of certain strings of symbols as "axioms" and certain rules, each of which allows the inference of a new string from certain previous ones. The strings which can be obtained from axioms by successive application of the rules are called theorems.

Logicism

The logicist approach, carrying on the tradition of Dedekind, Frege and Peano and of which Russell was one of the most famous exponents, held that mathematics should be reduced to logic and that the well-known paradoxes arise due to disregarding the types of concepts. Accepting this program involved taking some platonist assumptions as intuitively evident. In seeking to reduce arithmetic to logic, in part the logicists wanted to show that no appeal to sensible intuition was necessary in arithmetic, as had been claimed by John Stuart Mill and Kant. In

---

[3] $\vdash A \vee \neg A$.

order to try to reduce number theory to logic, the notion of a cardinal number was defined as the number of some class. Together with Poincaré, Russell thought that the cause of the paradoxes lay in the use of impredicative definitions[4]. Russell's solution to the paradox he invented was to use a hierarchy of types; similarly, a hierarchy of orders was used to avoid the semantic paradoxes (eg. the liar paradox and Weyl's paradox[5]) by excluding impredicative definitions. The resulting ramified theory of types necessitates postulation of the (contentious) axiom of reducibility, which states that any higher-order property or proposition could be replaced by some first-order one. Moreover, many terms have different senses for each type or order. Although logicists claim to reduce mathematics entirely to logic so that mathematical objects can be defined in logical terms via classes, and mathematical proofs can be reduced to logical proofs, the approach is generally deemed to have failed because, even if part of the essence of mathematics is not to do with intuitive understanding, set theory is arguably not part of logic: classes are not necessarily logical objects and Russell had to introduce new axioms (of infinity and reducibility) which are not part of logic. Even Russell was not entirely happy with the introduction of these, saying of the axiom of reducibility in the introduction to (Whitehead & Russell, 1925):

> "This axiom has a purely pragmatic justification: it leads to the desired results, and to no others (so far as is known). But clearly it is not the sort of axiom with which we can rest content."

Since these introduced axioms are not essential to reasoning, a reduction of arithmetic to set theory does little to increase the clarity of the foundations of arithmetic.

---

[4]An impredicative definition of an object is one which depends upon a set of which the object is a member.

[5]Namely whether "heterological" (ie. not applying to itself) is a heterological adjective.

Intuitionism

A further foundational position is represented by intuitionism. The belief that mathematics is invented has given rise to a controversial theory known as constructive mathematics, which is motivated by the belief that claims about the existence of mathematical objects and the truth of mathematical propositions should be judged according to the possibilities of construction of these objects and propositions. By "possibilities of construction" is meant "idealised possibility of construction", in order to take account of the limitations of human beings' abilities, and to cope with the problem of assertions about the infinite.

The intuitionists, led by L.E.J. Brouwer and Arend Heyting, developed a constructive system of logic which formed a coherent whole, differed from classical mathematics, and moreover thought of the mathematician, Brouwer's "creative subject", as being an integral part of the mathematical process. They believed that the subject-matter of mathematics is confined to what can be constructed by the mind. The name "intuitionism" is due to Brouwer's acceptance of Kant's claim that our concept of natural numbers is due to an *a priori* intuition of temporal succession (Bernays, 1983, P264):

> "What Brouwer appeals to is evidence. He claims that the basic ideas of intuitionism are given to us in an evident manner by pure intuition. In relying on this, he reveals his partial agreement with Kant. But whereas for Kant there exists a pure intuition with respect to space and time, Brouwer acknowledges only the intuition of time, from which, like Kant, he derives the intuition of number"

Thus, the intuitionists' use of intuition is in the sense derived from Kant, roughly meaning perception, and is (Heyting, 1934):

> "nothing other than the faculty of considering separately particular concepts and inferences which occur regularly in ordinary thinking."

According to Heyting, there is for mathematics (Heyting, 1934):

> "no other source than an intuition, which places its concepts and inferences before our eyes as immediately clear."

The philosophical ideas on which intuitionism is based go back at least to Aristotle's analysis of the notion of infinity as "the open possibility of more" which can therefore never be actualised, and his rejection of Plato's views about the absolute existence of mathematical entities and the "Complete and the Whole" notion of infinity ($\tau\acute{\epsilon}\lambda\epsilon\iota o\nu$ $\kappa\alpha\grave{\iota}\acute{\delta}\lambda o\nu$) (Wickstead & Comford, 1929, P206):

> "The infinite turns out to be the contrary of what it is said to be. It is not what has nothing outside it that is infinite, but what always has something outside it."

By the nineteenth century constructivist ideas were being clearly expounded by Leopold Kronecker, amongst others. However, intuitionism itself is generally acknowledged to have been founded by the Dutch mathematician Brouwer, who set out his ideas in several publications from 1907 onwards (Brouwer, 1964). This is in the main because he was the first person to assert a conclusion from which other influential contemporary writers with similar ideas, such as Poincaré, shrank: that constructivism should require a substantial modification of classical logic and analysis. Brouwer went beyond Kronecker's methods by the systematic application of forms of abstract reasoning and succeeded in establishing a general intuitionistic logic, which has been systematised by Heyting. Although mathematical intuitionism is associated especially with L.E.J. Brouwer and A. Heyting, there are other mathematicians with a similar viewpoint: for example, Borel and Lebesgue who are known as semi-intuitionists (or the French empiricists), and the constructivist Bishop (Bishop, 1967).

Brouwer's point of view leads immediately to a criticism of the laws of excluded middle and of negation. If we assert "$\neg A$", we are claiming that there is a proof of $\neg A$ (or, as Brouwer says, a construction exists which obtains an absurdity from the supposition of a proof of A), which is quite another matter from not having a proof of A. As illustrated by Fermat's Last Theorem[6] (or indeed any other as yet unsolved problem of mathematics), there does exist an A such that we are not in

[6]For all integers $n > 2$, and all positive integers x, y, z : $x^n + y^n \neq z^n$.

a position to assert A or $\neg A$, although either may ultimately be assertible. Hence, in this rather weak sense the law of excluded middle does not hold. The proof-theoretic result is that intuitionist logic accepts part of classical predicate logic (in fact it rejects the law of excluded middle, or, equivalently, the law of double negation[7]). In addition, and for the same reasons, the procedure of *reductio ad absurdum* is rejected. So, the intuitionist point of view leads to a distinctive logic and also to a distinctive theory of the foundations of analysis.

Hilbert's Programme

Another approach to the foundations of mathematics was taken by David Hilbert; his programme in the 1920s to prove that any problem that could be given a mathematically precise statement could be solved using logic was at the time widely viewed as the contemporary challenge facing mathematics. This viewpoint is sometimes referred to as formalism, although the designation is misleading, since Hilbert never said that platonist mathematics could be simply defined as a "meaningless" formal system. Hilbert thought that the logical paradoxes showed that non-finitary mathematics needed justifying, and the objective of his programme was to secure the foundations of Platonist mathematics by a finitary[8] analysis of classical formal systems. His method was to axiomatise classical mathematics and prove the consistency of the resulting formalism by finitary means within meta-mathematics (also called proof theory), which is a metatheory in which properties of the formal mathematical system are described and studied. He hoped to formalise all of mathematics (or at least, analysis) into a single system, and that the consistency of this system could be proved by methods so elementary that no-one could question them.

---

[7] $\vdash \neg\neg A \rightarrow A$.

[8] Finitism is a method rather than a theory, and is a form of constructivism which insists that operations must be done in a finite number of steps with a finite number of elements. Strict finitism is an extreme form which insists that constructions must be carried out in practice.

However, Gödel's incompleteness theorem (Gödel, 1931)[9], plus Skolem's theorem that it is not possible to characterise the system of natural numbers by a countable system of axioms stated in the predicate logic of the first order (Skolem, 1934), show that there is a certain limit to the effectiveness of Hilbert's method, and that formal logical methods do not allow all arithmetical questions to be answered. Indeed, it is not certain that strong enough constructive methods can be found even to prove the consistency of classical analysis: in 1963 Kreisel showed that intuitionist analysis, with the bar theorem and a strong schema of "generalised inductive definitions" included, does not suffice to prove the consistency of classical analysis (Kreisel, 1965).

Work on Hilbert's programme has nonetheless continued, with contributions by Ackermann, von Neumann, Skolem, Herbrand, Gödel and Gentzen. However, it was necessary to work with formalisms that embody only part of mathematics, and proofs which rely on more abstract notions — but nevertheless valuable results were achieved. For example, Gödel and Gentzen proved independently (and finitistically) that if intuitionistic first-order arithmetic is consistent, then so is classical first-order arithmetic. The proofs were based on a method of translating classical theories into intuitionist theories cf. (Baker, 1989). Nonetheless, these investigations have remained within a relatively restricted domain, and have not even enabled a proof of the consistency of the axiomatic theory of integers.

Further Viewpoints, and the Formalisability of Arithmetic

Contrasting notions of the nature of arithmetic have already been presented. In addition, there has been more modern interest and discussion about its nature, such as Penrose's argument that our insight into arithmetic is non-algorithmic in nature (Penrose, 1989), or Lakatos' notion of the evolution of mathematics

[9]See earlier discussion in this section. Gödel's theorem (incorporating strengthenings due to Rosser and Tarski, Motowski and Robinson) shows that, in any consistent axiomatisable extension of certain finitely axiomatisable subtheories of Peano arithmetic, there are propositions of number theory that can neither be proved nor disproved.

(Lakatos, 1976). Penrose, amongst others, has argued that it is mathematical insight which allows us to see the truth of true but unprovable statements, and hence that this notion of mathematical insight cannot be completely captured by any algorithm. He writes (Penrose, 1989, P141):

> "When we convince ourselves of the validity of Gödel's theorem we not only 'see' it, but by so doing we reveal the very non-algorithmic nature of the 'seeing' process itself ... Real mathematical truth goes beyond mere man-made constructions."

On the other hand, Dummett holds that provability is not a stable property and that indeed mathematics is a subject whose results are liable to revision, since individual sentences have meaning (Dummett, 1977, P403):[10]

> "The meanings of our mathematical statements are always, to some degree, subject to fluctuation."

It is not appropriate to explore these viewpoints in depth at this stage, but merely to note that influential developments of the foundational positions given previously have been put forward; Penrose's viewpoint in particular shall be discussed further in Section 2.7.

In this subsection the foundational approaches of logicism, intuitionism and Hilbert's approach, together with formalism and platonism have been characterised. Of these positions, only intuitionism, the classical (in the sense of platonic) approach and the finitary approach are serious contenders as relevant methods to be used for clarification of the nature of the ω-rule. In the following subsections the notion of canonical proofs is discussed, and also various notions of proof, infinite structure and infinite proof which have been proposed are presented, in order to gain some philosophical insight into the nature of the ω-rule.

2.1.2 Distinctions Relating to the Idea of a Proof

In this subsection various notions of proof will be considered.

---

[10]For further details of this discussion, see (Prawitz, 1977).

Intuitive Proofs Versus Formal Proofs

Contrasting with the formalist viewpoint of proofs as uninterpreted linguistic structures (ie. unjustified symbols), according to Hilbert's approach it is the case that discussion about such a formal system (which takes place within some metatheory and may indeed itself be considered part of mathematics) must of course be meaningful. The platonic view of proofs as interpreted linguistic structures, and as meaningful symbols which are an operation serving to test or check the correctness of an arithmetical calculation, accords with the logicist view that mathematics deals with structures in a formal manner, independently of their meanings. These viewpoints have in mind the idea of formal proofs, which are proofs in a formal system (Kleene, 1962, P83):

> "A *(formal) proof* is a finite sequence of one or more (occurrences of) formulas such that each formula of the sequence is either an axiom or an immediate consequence of preceding formulas of the sequence."

In this sense, the proof of $\forall x\ x \rightarrow x$ will be some derivation in a system, such as:

$$\dfrac{\dfrac{\dfrac{}{r \vdash r}\,axiom}{\vdash r \rightarrow r} \rightarrow -r}{\vdash \forall x\ x \rightarrow x} \forall - r$$

On the other hand, the intuitionist position is that proofs are mental: Brouwer insisted that proofs are (possibly infinite) mental constructions (Heijenoort, 1967, P64):

> "These *mental* mathematical proofs that in general contain infinitely many terms must not be confused with their linguistic accompaniments, which are finite and necessarily inadequate, hence do not belong to mathematics."

and (Dalen, 1981, P4):

> "... intuitionistic mathematics is an essentially languageless activity of the mind".

Dummett writes (Dummett, 1977):

> "Mathematical objects, unlike concrete ones, can be apprehended only in thought; hence if they are not regarded as themselves the products of thought, that can be only because they are viewed as the permanent possibilities of certain mental operations."

Intuitionism insists that in mathematics we should assert a mathematical statement only if we either know it by immediate intuition or know how to prove or disprove it, using steps known by intuition. A proof known by intuition is just one which is evident to the mathematician and an acceptable proof of something's existence involves constructing it. Intuitionists do not believe, as formalists do, that mathematics is to be found in formal systems, but on the other hand they do recognise formalisation as an important tool for mathematical investigation. The mental constructions exist in the mind of an idealised mathematician. The language of mathematics is an (inadequate) attempt to describe these mental constructions.

Brouwer's position is that no mathematical proposition is true unless we can in some nonmiraculous way *know* it to be true. True to this notion, any mathematical object will exist only if it can be constructed. Hence, whenever a constructive proof mentions the existence of an object, it must provide a method of "finding" that object. So, proving $\exists x F(x)$ means showing how to construct an x such that $F(x)$, and proving $\forall y \exists x F(x, y)$ must involve giving a general method for finding x on the basis of y. The logical connectives are interpreted in such a way that a disjunction A or B constitutes for the intuitionist an incomplete communication of a statement which either says that A holds, or that B holds, or else gives a method by which one may choose from A or B one which holds.

Thus, an intuitive proof is a mental construction giving a justification: it makes something *evident.* This encapsulates a Wittgensteinean approach of "we recognise a proof when we see one" (Sundholm, 1983a, P156). It is a primitive notion which may be used in giving more complex justifications (for example that the proof of $A \wedge B$ is comprised of the pair of the proof of A and the proof of B). Heyting's "Interpretation of the Signs" (Heyting, 1956, P96–7) is such a process, when he defines the semantics of the intuitionistic connectives: "$p \wedge q$ can be asserted if

and only if both of p and q be asserted" etc., for which a summary may be given (Kreisel, 1962b, P201–202):[11]

> "As Heyting indicated, one then obtains proof conditions for A ... In particular, if A is $B \circ C$, where $\circ$ denotes a logical connective and r_B and r_C determine the sense of B and C, then we obtain r_A from r_B and r_C."

In the above explanations the meaning of a proposition is given by its 'proof condition' and (as Kreisel stressed in (Kreisel, 1962b)) in some sense proofs are evident in that they are recognisable as such. Heyting thought that a mathematical proposition is proved by carrying out a certain construction (which must satisfy certain properties such as giving a mathematical object). The proof objects are in the sense of his definition of proof of $p \rightarrow q$ in the "Interpretation of the Signs" that (Heyting, 1956, P97):

> "The *implication* $p \rightarrow q$ can be asserted, if and only if we possess a construction r, which, joined to any construction proving p (supposing the latter be effected), would automatically effect a construction proving q."

Such functions are proof objects: Heyting considered that there were two primitives for meaning of the basic logical notions, namely judgement or proof (especially of identities), and functions. Pairs of proofs p and functions Π of the form (p, Π) would be defined in the sense that '(p_0, Π_0) *establishes* $A \rightarrow B$' means that p_0 is a proof of[12]

$$[(p, \Pi) \textit{ establishes } A] \Rightarrow [\Pi_0(p, \Pi) \textit{ establishes } B] \quad \forall p, \Pi$$

A possibility introduced later by Curry and Howard is that proofs as mental constructs can be approximated by λ-terms whose execution corresponds to some

---

[11]This relates to the field of "compositional semantics" — see for example (Gaifman & Shapiro, 1988).

[12]However, any such list of formal laws leaves some room for different interpretations (Kreisel, 1971, P143).

such construction (Howard, 1980). This resulting proof object can be thought of as a constructive witness term. Thus, if what might constitute a proof in type theory of, say, $\forall x : U\ x \rightarrow x$ (where $x : U$ means that the object x is of type U) is considered, for this approach $\lambda a.\lambda b\ .\ b$ will be a proof. Similarly, Kreisel used '$\Pi(c, A)$' for the relation "construction c proves proposition A" (Sundholm, 1983a, P154). Note how Heyting's position (that a mathematical proof is of the form of a process of construction carried out to produce a mathematical object, thereby proving a proposition) relates to the idea of a proof presented in terms of a function generating a well-founded and correct tree of sequents (cf. Chapter 5).

Sundholm differentiates between a construction as a process of construction (ie. making a proof evident), the object obtained as the result of a process of construction or the construction-process as an object (Sundholm, 1983a, P164). Brouwer's notion of construction is quite different from Heyting's, for Brouwer believed that within mathematical activity (processes of) constructions are carried out, but when reflection on these happens, the processes may be treated as objects (Sundholm, 1983a).

There is also the notion of proof as the completion of a task to be considered, for Heyting considered that mathematical proof is the fulfillment of the intention expressed by a mathematical statement. Sundholm shows that this is essentially the same as what is known as the Kolmogorov explanation, which states that a mathematical statement is to be viewed as the expression of a problem, and so the intention expressed by the statement is fulfilled if and only if the problem is solved (Sundholm, 1983a). So, with reference to the ω-rule: 'If every number n has the property $A(n)$, then $\forall x A(x)$ holds', the intuitionist interpretation would be that if it is proved that every number n possesses the property $A(n)$, then $\forall x A(x)$ holds, whereas the sense of Kolmogrov's interpretation of intuitionism would be the claim that there exists a proof (which would be effectively given) of $\forall x A(x)$ from the given hypothesis. The former is the approach taken with regard to the implementation (cf. Chapter 4) and the 'general proofs' of Chapter 6 provide the justification that every number n has the property $A(n)$.

In conclusion, the distinction between formal and intuitive proofs corresponds to the differing views of proof given by formalists and intuitionists, both of which may be useful.

Canonical Proofs Versus Demonstrations

Within the intuitionist position, there is a further differentiation between canonical as compared with non-canonical proofs. Dummett highlights the distinction between constructions which are supposed to constitute proofs of mathematical statements (in the sense in which 'proof' is used in the intuitive explanations of the logical constants mentioned above, which are mental constructions) as opposed to so-called 'demonstrations' (Dummett, 1977). The primary notion is that of a proof in the strict sense, which Dummett calls a *canonical* proof; the notion of demonstration is secondary, and definable in terms of this. A demonstration is considered to supply (at least in principle) an effective method of constructing an actual proof. The type of explanations provided in textbooks are demonstrations rather than canonical proofs. So a demonstration (or non-canonical proof) may be thought of in terms of being an indication of how to construct a rigorous (ie. canonical) proof — the latter is not actually being carried out, but it would be possible to do so. The notion of non-canonical proof may also correspond to the metatheory: a metamathematical proof is an intuitive demonstration of the truth of a meaningful statement about the formal objects of a theory (Kleene, 1962, P85).

There are several reasons for the canonical versus non-canonical proof distinction. First, it avoids circularity in explaining the logical constants; this is necessary in defining what counts as a canonical proof of $A \rightarrow B$, say, where proofs already involving modus ponens should be disallowed (at least for those containing $A \rightarrow B$ as a premise). Arguments involving modus ponens may be valid, but would be demonstrations rather than canonical proofs. The same applies for universal quantification with regard to universal instantiation, and negation with regard to *ex falso quodlibet* (Dummett, 1975). Heyting wrote about this notion of circularity (Heyting, 1934):

> "It would be circular to apply any philosophical or logical principles as means of proof, since mathematical conceptions are already presupposed in the formulation of such principles."

In the sense that is given to 'proof' in the explanations of the logical constants given by Heyting's "Interpretation of the Signs", the informal proof (which might contain the line $A \vee B$ because there is an effective method of obtaining such a proof for example) does not actually provide a proof of the disjunctive statement — it provides only a method, effective in principle, for finding such a proof. When we have to decide whether to accept a given construction as a proof, say of a conditional statement $A \rightarrow B$, we need to judge whether, when applied to an arbitrary proof of A, it will yield a proof of B. In so doing, we should consider only those possible proofs of A that have a complexity below a certain bound determined by the complexity of A, and hence of lower complexity than is required for a proof of $A \rightarrow B$ itself. This argument applies to the other logical constants in a similar manner.

Secondly, an important result concerning pure logic obtained in finitist metamathematics is provided by theorems such as Herbrand's theorem (1931) and Gentzen's theorem (1934), to the effect that the proof of a formula of first-order logic can be put into a normal form. In such a normal form proof the logical complexity of the formulae occurring in the proof is in certain ways limited in relation to the complexity of the conclusion; for instance, no formula can contain more nested quantifiers than the conclusion, and the cut rule is eliminated (cf. Section 2.3). As a consequence, a quantifier-free formula deduced from quantifier-free axioms can be proved by propositional logic and substitution. Normalised natural deduction proofs provide an exact analogy for what is required of canonical proofs if the intuitionistic explanations of the logical constants are to form a system free of conceptual circularity. It does not follow that this holds for any formalised first-order theory — the comparison between canonical proofs and normalised proofs only holds in a natural deduction system. Thus, the conversion to canonical proofs may be regarded as a verification procedure. More specifically, the deletion of elimination rules from a canonical proof suggests a comparison with the notion

of a normalised deduction, for normalisability implies that for each logical constant, the full language is a conservative extension of that obtained by omitting the constant from its vocabulary (Dummett, 1975).

Thirdly, the distinction between a canonical and a non-canonical proof provides a constraint on a logic. For example, if elimination rules and introduction rules are laid down, one might wish to establish some sense in which the latter preserve "soundness". Note that a non-canonical proof does not fix meaning, whereas a canonical one does: so, "there exists a canonical proof" may be taken to be the intuitionistic analogue of "true". Dummett writes (Dummett, 1975, P15):

> "We must, therefore, replace the notion of truth, as the central notion of the theory of meaning for mathematical statements, by the notion of *proof*: a grasp of the meaning of a statement consists in a capacity to recognise a proof of it when one is presented to us, and a grasp of the meaning of any expression smaller than a sentence must consist in a knowledge of the way in which its presence in a sentence contributes to determining what is to count as a proof of that sentence."

From the intuitionistic viewpoint, arithmetical statements are understood in terms of their assertibility, rather than their truth. Does such assertibility relate to the existence of a canonical proof or of a demonstration? Dummett claims that a statement is assertible provided that we are in fact in possession of (an effective) means of obtaining a canonical proof of it, whether or not we are aware of the fact (for example, we may not be aware of the fact in the cases of disjunctive or existential statements) (Dummett, 1975). So an (indirect) proof of a sentence would be a method for obtaining a direct (or canonical) proof, and the meaning of the sentence is given by this canonical proof. In other words, the claim is that the point of reasoning is to establish something indirectly which in some sense could have been established directly. Constraints are needed for how the proof-conditions of Heyting's constructive proof-tables (ie. the "Interpretation of the Signs") are to be understood: Heyting's explanation of logical constants roughly corresponds to taking the introduction rules as fixing the meaning of the logical constants (Heyting, 1956). So, the introduction rules for a constant represent the direct or canonical means of establishing the truth of a sentence with the constant as principal operator; they are synthesising rules, explaining how a proof of $A \circ B$

can be formed in terms of the (given) proofs of A and of B. Then, the elimination rules are justified by reference to the introduction rules (Dummett, 1991, P252). In fact, the idea is to have either the introduction or the elimination rules serving as the direct, meaning-giving way of learning the truth, with the other providing the indirect means.

Fourthly, demonstrations are a useful distinction because sometimes one might want to regard something as demonstrated due to the existence of an effective method for constructing a proof (for example that some huge number either is or is not prime) — that is to say that it is possible in some sense to give a proof.

Finally, this distinction between canonical and non-canonical proofs is intuitively plausible. Sundholm considers the statement "I have 7 coins in my pocket"; this may be verified by counting or calculating, and is not likely to be verified in a wholly formal manner (Sundholm, 1986, P487):

> "In most cases the use of valid inference has very little to do with how one would normally set about to verify the truth of something. For instance, the claim that I have seven coins in my pocket is best established by means of counting them. It would be possible, however, to deduce this fact from a number of diverse premises and some axioms of arithmetic. ... This would be a hightly *indirect* way in comparison with the straightforward counting process. The utility of logical reasoning lies in that it provides indirect means of learning the truth of statements. Thus in order to account for this usefulness it seems that there must be a gap between the most direct ways of learning the truth and the indirect ways provided by logic. If we now explain meaning in terms of proof, it seems that we close this gap. The direct means, given directly by the meaning, would coincide, so to speak, with the indirect means of reasoning. ... one must not make the identification between proof and meaning so tight that logic becomes useless."

In this subsection a useful distinction between canonical and non-canonical proofs has been given, together with motivations for so doing. Canonical proofs are rigorous proofs which constitute the meaning of a proof.

In the case of the ω-rule, the ω-proofs are the canonical proofs of universally quantified statements over the natural numbers, to the effect that, given a natural number, a proof is returned of that instance.

2.1.3 Notions of Infinite Structure

This subsection is concerned with investigating the widely differing views held about the nature of infinite structures, with a view to next considering infinite proofs.

The classical viewpoint (encompassing Dedekind and Cantor) is of a platonic mathematical structure. Cantor's theory has been called a theory of the actual infinite since he is said to have thought of infinite classes as actual or completed. The infinite is actual or completed, in the sense that an infinite set is regarded as existing as a completed totality, prior to or independently of any human process of generation or construction, as though it might be seen in its entirety. So, classical mathematics views an infinite process rather as if it were a particularly long finite one. Universal propositions are said to have definite truth-values. Classical analysis also accepts the totality of subsets of the natural numbers, which allows deduction of theorems such as that any real number is either equal to 0 or not. Bernays argues that we can regard the totality of sets of natural numbers as analogous to the totality of subsets of a finite set. An "arbitrary" subset of the natural numbers is fixed by an *infinity* of independent determinations fixing for each natural number whether it belongs to the subset or not (Bernays, 1983, P260):

> "In the same way, one views a set of integers as the result of infinitely many independent acts deciding for each number whether it should be included or excluded. We add to this the idea of the totality of these sets."

This assumption of the totality of arbitrary subsets of the natural numbers justifies impredicative definitions. Weyl proposed to construct analysis on the basis of platonism merely with respect to the natural numbers, and managed to reconstruct a large part of analysis without using impredicative definitions.

In a related way, the logicists' postulation of the "axiom of infinity" implies infinite totalities, and the existence of the natural number series as a hypothesis about the actual world.

The formalist view of infinite structures is that they cannot exist (within a formal theory), in the sense that the objects of interest are finite linguistic structures.

Hilbert, with his metamathematics, thought that no infinite class may be regarded as a completed whole, and that proofs of existence should give at least implicitly a method for constructing the object which is being proved to exist.

The intuitionists' view is that nothing more can be known about infinite structures than may be derived from their generating rules. Intuitionists refuse to assume that infinite sets exist, although allowing rules which generate ever larger finite sets. The infinite is potential or becoming or constructive, rather than actual. The natural numbers are regarded as mental constructions, generated in a way which may be determined by repeated application of the successor function to 0.

An early challenge to the classicists' view along these lines was by Gauss, 1831 (Kleene, 1962, P48):

> "I protest ... against the use of an infinite magnitude as something completed, which is never permissible in mathematics."

Principles valid in thinking about finite sets do not necessarily carry over to infinite sets (for example that the set of natural numbers contains a greatest member). Note also that Brouwer did not accept the law of excluded middle for infinite sets, but only for finite ones (see (Kleene, 1962, P47)).

So, for the intuitionists, an infinite structure should be thought of as something which is always in the process of being constructed. In the case of the ω-rule, a proof of $\forall x A(x)$ will be a process which is seen to result in a proof of $A(\underline{n})$ for any number n which is generated (cf. Subsection 2.1.2). To conceive of an infinite structure is to understand the (non-terminating) process which generates it. As Dummett says (Dummett, 1977, Chapter 3),

> "... all that we can, at any given time, know of the output of the process of generation is some finite initial segment of the structure being generated. There is no sense in which we can have any conception of this structure as a whole save by knowing the process of generation."

In other words, each individual member of an infinite sequence may be constructed, but there is no construction which contains within itself the whole infinite sequence.

2.1.4 Views of Infinite Proofs

I shall now consider the approach of various factions regarding infinite proofs, and hence the use of ω-rules. In particular, the question of whether infinite proofs are mathematical objects in the same sense as finite proofs is considered.

The formalist view is that proofs about infinite structures may be provided via axiomatisation, for the formalisation of an axiomatic system can give rise to an infinite system — for example, if all instances of a schema are taken as axioms. There must however be a mechanical procedure for deciding whether a given formula is an axiom and whether a given inference of a formula from finitely many premises is correct according to the rules of inference.

Zermelo, although not an intuitionist, agrees with the intuitionist position that we try to understand infinite proofs by approximation via iterations of basic steps, rather than as a whole (Zermelo, 1932, P85):

> "The true subject-matter of mathematics is not, as many believe, 'configurations of symbols', but *ideal abstract relations* between the elements of a conceptually established *infinite manifold*, and our systems of signs are thereby only auxiliary devices, always incomplete and changing from case to case, in our attempts to achieve step-wise mastery of the infinite, which we cannot immediately and intuitively 'survey' and 'comprehend'."

The intuitionistic view is that infinite proofs are seen in terms of generating principles. For Brouwer's notion of a proof of the ω-rule, the appearance of a universally quantified statement in a proof $(\forall x A(x))$ will just signify our recognition that a certain operation when applied to any element of the domain will yield a proof of the corresponding instance, that is $A(\underline{k})$ in the natural number domain. In the fully analysed proof the universal quantification line is essentially left out, and left implicit. Dummett writes (Dummett, 1975):

> "a proof of a universally quantified statement will be an operation which, applied to each natural number, will yield a proof of the corresponding instance; and if this operation is carried out for each natural number, we shall have proofs of denumerably many statements. The conception of the mental construction which is the fully analysed proof as being an infinite structure must, of course, be interpreted in the light of the intuitionistic view that all infinity is potential infinity: the mental construction consists of a grasp of general principles according to which any finite segment of the proof could be explicitly constructed. ... Indeed, it might reasonably be said that the standard intuitionistic meanings of the universal and conditional quantifiers involve that a proof is such a potentially infinite structure."

and

> "A proof containing inferences with infinitely many premises appears non-constructive only because we try to imagine ourselves as completing the infinite process of proving each of the premises individually, and then drawing the conclusion, and this, of course, makes no constructive sense. But, on the intuitionistic understanding of infinity, the only way in which we can draw an inference from infinitely many premises is by recognizing *that* each of these premises can be proved; and that, in turn, can be accomplished only by recognizing, of some general procedure, that it will yield a proof of each of the premises. Thus the only way of understanding the idea of an inference from denumerably many premises $A(0), A(1) \ldots$ which is consistent with a constructivist outlook proves to coincide exactly with the intuitionistic interpretation of an inference from $\forall n A(n)$. An intuitionistic proof involving inferences from universally quantified statements really is, therefore, what Brouwer maintains, a representation of a more fully analyzed proof containing inferences from infinitely many premises."

Therefore, an ω-rule proof is a *canonical* proof of an arithmetical universal statement. Note how the general procedure mentioned above corresponds to the generating function *gen_pf* (which generates individual proofs) given in Chapter 6.

So, a proof containing inference rules which have infinitely many premises, such as the ω-rule, cannot be a written proof, but could be a mental construction, or from the classical approach, a platonic structure. The proof must however be well-founded (so if represented in tree form, every branch must be finite). However, it is not necessary that every proposition occurring in the proof should depend on only finitely many premises. Indeed, this is the form of the representation of the system incorporating the constructive ω-rule presented in Chapter 4. Dummett

summarises the intuitionists' view of infinite proofs as follows (Dummett, 1977, P95–6):

> "At first sight, this appeal to the conception of an infinite, though well-founded, proof is contrary to the constructivist spirit. However, we must interpret it in accordance with the general intuitionistic doctrine about the nature of infinity: an infinite structure can never be regarded as completed; rather, to grasp an infinite structure is to grasp a principle whereby any finite segment of it, however large, can be explicitly constructed. Looked at in this light, it can hardly be denied that an intuitionistic proof supplies a principle for the explicit construction of any finite segment of its fully analysed version, so long as it is understood that the direction of analysis, and thus of construction, runs counter to the direction of inference."

The formal theory of the ω-derivations given in Chapters 4 and 5 provides an attempt to describe infinite mental structures, generated in a primitive recursive way, and analysed as well-founded trees with basic axioms at the leaves.

2.1.5 Conclusions Regarding Foundational Positions

For even non-advocates, intuitionism provides a new way of looking at mathematics and logic, and a productive challenge to the old order. Moreover, its ideas have ramifications for many-valued truth systems, and hence in such diverse fields as the philosophy of possible worlds and electronics. Yet classical logic remains a useful, well-developed model, in many cases less cumbersome than intuitionist logic, and so has not been replaced. From a purely practical point of view, there is a place for both systems, each more suitable for certain contexts and having differing benefits and disadvantages. By its nature, intuitionist logic produces useful algorithms, while classical logic is able to "prove" more sequents and gives more concise proofs — complexity of proofs is weighed against the provision of a ready algorithm. The most interesting aspects of the intuitionistic approach regarding proofs from my point of view are that they should be constructive, and also be convincing (in the sense of being correct). The intuitionistic attitude towards relevance of intuition in the mathematical process has no practical consequences here.

In conclusion, the implementation of the constructive ω-rule described in Chapter 4 embodies an intuitionistic approach to the representation of an infinite proof system. The classical alternative (the full ω-rule) is not suitable for implementation. Classical mathematics would not object to the use of a constructive ω-rule, for some people who object to intuitionism are quite happy with the classical restriction to recursive functions. Although the classical viewpoint is not as restricted as the intuitionistic one, the constructive restriction on infinite proofs does make sense.

So, my position regarding infinite proofs shall be the intuitionistic one in so far as I regard the generating process as sufficient to characterise them. This ties in well when one attempts to represent such proofs in a finite way, as is necessary for the implementation of a system involving infinitary rules such as the ω-rule. Primitive recursive well-founded trees shall be presented with respect to a generating function, namely f, defined in Section 5.3. Note the correspondence between the notion of intuitive proofs (as constructions giving objects) and the idea of a proof presented in terms of a function generating a well-founded and correct tree of sequents, together with the relation between formal proofs and the description of the semi-formal system $PA_{c\omega}$[13] given in Chapter 5.

In addition, I shall conclude that canonical proofs within $PA_{c\omega}$ provide a notion of soundness for this system, in the sense described above. A canonical proof of a statement S within the system $PA_{c\omega}$ will be provided by a well-founded and correct higher order primitive recursive tree with S as the sequent at its topnode, as shall be defined in Chapter 5. As mentioned in the previous subsection, the ω-rule proof is equivalent to a canonical proof of, and thus provides a meaning for, $\forall n \in nat\ \phi(n)$.[14] The implementational representation of general proofs (using

---

[13] $PA_{c\omega}$ is a representation of arithmetic with a primitive recursively restricted ω-rule in place of the rule of induction; see Section 4.3 for further details.

[14] Note also that the numerals in the hypotheses of the rule are themselves canonical representations (of numbers).

rewrite rules) is a matter of convenience and clarification, and corresponds to 'demonstrations'.

As a side issue, there is a difference of opinion about the shape of constructions which prove a universal sentence $\forall x A(x)$, which is analogous to the preference for reflection systems (see Section 11.1) versus ω-rules. Kreisel and Troelstra advocate that it should be pairs of the form (e, f), where $f(n)$ is a proof of $A(\underline{n})$, and e proves that f has this property. On the other hand, Dummett, Prawitz and Martin-Löf believe that the function f alone is enough (Sundholm, 1978). In my representation, I have taken the latter view, and have not included a proof of the fact that the function generates a prooftree with $A(\underline{n})$ as root.

2.2 Historical Development of the Omega Rule

Tarski and Hilbert seem to have both formulated early versions of the ω-rule independently. Apparently, the earliest proposal of an ω-rule was by Tarski (Tarski, 1936, P279, note 2), who claimed that he introduced the rule in an "unpublished talk" to the Second Conference of the Polish Philosophical Society at Warsaw in 1927, although the first publication in which this was mentioned is (Tarski, 1933). He calls the rule by the name of "the rule of infinite induction", but it is in fact what we now call the ω-rule. However, Hilbert is often credited with introducing the rule, which he named "transfinite induction". In particular, lines 13–18 on P491 of (Hilbert, 1930) read as follows: "If it has been demonstrated that the formula $A(z)$ is true for every given numeral, then the formula $\forall x A(x)$ may be used as an axiom." The rule was then formulated again (as the rule $DC2$: $\forall x A(x)$ is a consequence of the collection of formulae $A(\underline{k})$) in (Carnap, 1934, Section 14), which led to the ω-rule being referred to as "Carnap's rule" in (Rosser, 1937), although this naming has not been followed. It seems likely that Carnap did formulate the rule independently from both Hilbert and Tarski (Isaacson, 1992, P12–14). Sundholm (Sundholm, 1978, P3) considers Hilbert's rule to be merely a

version of the rule of uniform reflection[15], with "demonstrated" to be interpreted in terms of provability, whereas it could instead be regarded as a constructive version of the ω-rule. In any case, the uniform reflection principle is a sort of formalised ω-rule, a name under which it was in fact known for some time, such as by Kreisel (Kreisel, 1962a) and Feferman (Feferman, 1960). The first published discussion of the formalised ω-rule was in (Rosser, 1937, PP134–5), where Kleene is credited with the idea. Turing's work on ordinal logics (Turing, 1939) is also highly relevant, and was developed by Feferman to form his work on progressions based on reflection principles (Feferman, 1962). Apparently, the name "omega-rule" did not occur until about 30 years after the rule was first formulated (Sundholm, 1978, P3), with (Grzegorczyk *et al*, 1958) being the first authors to actually use the name "rule-ω". The closely related methods for dealing with incomplete systems, namely adding reflection principles or adding restricted ω-rules,[16] essentially led to the same result: that every true arithmetical sentence is provable. Emerging from these various rules, as shall be seen in the following sections, one of the most promising options as regards implementation is the *constructive ω-rule.* Informally, this may be described as the following: "If each $P(n)$ can be proved in a uniform way (from parameter n), then conclude $\forall n P(n)$."

2.3 The Importance of Cut Elimination

The cut elimination theorem is an important proof-theoretic result, which may be automated. It states that if a sequent $\Gamma \vdash \Delta$ is provable in classical or intuitionistic calculus respectively, then one can find a cut-free proof of $\Gamma \vdash \Delta$ in classical or intuitionistic calculus respectively. (That is to say that a proof for the same sequent may be found, in which the 'cut' rule of inference is never used.) The

[15]See Section 11.1

[16]See Sections 11.1 and 2.5 respectively.

following sections provide first theoretical background and then discussion about cut elimination within theorem proving, which provides a motivation for my work.

2.3.1 The Cut Rule

The cut rule is a very significant rule in the sequent calculus presentation, because it is the only way of inserting formulae into the proof which are not subformulae[17] of the current sequent. Several cut rules exist[18]; probably the most widely used is the classical cut rule:

$$\frac{\Gamma \vdash A, \Delta \quad \Psi, A \vdash \Pi}{\Gamma, \Psi \vdash \Delta, \Pi} \tag{2.1}$$

where Γ, Π, Δ and Ψ are arbitrary lists of formulae, and A is an arbitrary formula. The left-hand premise is of the generalisation, and the right-hand premise is a justification that it is indeed a generalisation, ie. that its addition to the hypotheses results in a proof of the original sequent. A is referred to as the *cut formula*, since it is a formula which is "cut in"; it is new in terms of top-down proof search. The intuitionistic cut rule which is normally used is:

$$\frac{\Gamma \vdash A \quad \Delta, A \vdash C}{\Gamma, \Delta \vdash C} \tag{2.2}$$

The cut rule is important in systems where cut is not a basic rule, such as natural deduction systems (for further details see Subsection 3.1.1). In this case it would be written as:

$$\frac{\Gamma, \Phi \quad \Gamma, \neg\Phi}{\Gamma}$$

2.3.2 Proof-Theoretic Rôle of Cut Elimination

Cut elimination theorems are widely regarded by logicians as being extremely important. In a general sense, they may be regarded in a way as tests of consistency,

---

[17]Subformulae are defined in a recursive manner, as according to Table 2–1.

[18]Texts such as (Schwichtenberg, 1977) and (Dummett, 1977) include the more common variants.

expression	*subformulae*
ϕ atomic	ϕ
$\neg\psi$	$\{sub(\psi)\} \cup \{\neg\psi\}$
$\phi \equiv \psi \wedge \theta$; $\phi \equiv \psi \vee \theta$; $\phi \equiv \psi \rightarrow \theta$	$\{sub(\psi)\} \cup \{sub(\theta)\} \cup \{\phi\}$
$\phi \equiv \forall x\ \psi(x)$ $\phi \equiv \exists x\ \psi(x)$	$\{\phi\} \cup \{sub(\psi(t))\}$ for t any term free for x in ψ

Table 2–1: Subformulae

for if a cut elimination theorem can be proved for some formalised theory of arithmetic, then that theory is consistent.[19] Another point which should be mentioned is that the cut elimination theorem of a particular system would enable the proof of the subformula property, which states that in any proof in that system of the sequent $\Gamma \vdash A$, every formula which occurs in any sequent in the proof must be a subformula either of A or of one of the formulae in Γ.[20] The subformula property enables a much firmer idea of the nature of the proof of a sequent in that system, since we are guaranteed that it is not the type of system where a formula may completely disappear from subsequent sequents (using a top-down approach, and analogously for bottom-up).

One particular relevance of the cut elimination theorem for predicate calculus is that information may be extracted from proofs. For example, there is the so-called existence property, the intuitionistic case of which states that if $\vdash \exists x_1, \ldots, x_n P(x_1, \ldots, x_n)$, with P quantifier-free, then the normal form of the proof yields terms $t_1, \ldots, t_n$ such that there is a proof of the statement $P(t_1, \ldots, t_n)$. This constructive version of Herbrand's Theorem is derived from the Cut Elimi-

---

[19] " ... if a cut elimination theorem can be proved for some formalised theory of arithmetic, it will follow that only numerical equations can occur in a proof of a numerical equation, and hence that there can be no proof of 0=1; and from this the consistency of the theory follows." (Dummett, 1977, P45)

[20] Subformulae are defined recursively, as shown in Table 2–1.

nation Theorem. It gives the (perhaps surprising) result that, upon the assertion (ie. proof, in intuitionistic terms) of the existence of something with a certain property, some object possessing that property will be specified. For classical logic, Herbrand's Theorem demonstrates that the provability of a quantifier-free formula of predicate calculus is consequent only on the language of the formula.

In addition, the cut elimination theorem is involved in the proofs of other important theorems, such as Beth's Theorem. This states that if a relation is implicitly definable (ie. in terms of itself) then it must also be explicitly definable (ie. only with regard to some external parameter). So, for example, in Peano arithmetic, $<$ may be defined (implicitly) via the axioms:

$$\forall x \neg (x < 0)$$

$$\forall x \forall y (x < Sy \leftrightarrow x < y \ \ or \ \ x = y)$$

An explicit definition of $<$ would be, for example,

$$x < y \leftrightarrow \exists z (z \neq 0 \wedge y = x + z).$$

This example is to illustrate the nature of implicit and explicit definitions — in fact, cut elimination is not true for arithmetic, hence Beth's Theorem does not hold for this system, and so those arithmetic relations definable by implicit and explicit means would not necessarily coincide.

2.3.3 Justification for Use of the Cut Rule

Why, then, do we have the cut rule? Well, as already mentioned, cut elimination does not hold for many systems, such as the usual axiomatisation of arithmetic. And although cut elimination is desirable for theorem-proving reasons (as discussed in the following subsection), in many cases use of the cut rule may lessen the length and obscurity of a proof, thereby making it easier to follow due to more "natural" proof steps. Moreover, the rule justifies the use of lemmata. There is some discrepancy as to whether a lemma is supposed not to be proved at the stage of introduction, as according to the definition given by Proclus (commentary on Euclid) that they are useful assumptions which need justification:

> "When, in either construction or demonstration, we assume anything which has not been proved but which requires argument, then we regard what has been assumed as doubtful in itself and worthy of investigation, and call it a lemma"

or whether lemmata enable repeated use of pre-proved information. Either type may be useful: the former for enabling proofs, and the latter for condensing proofs, for example. In general, lemmata can clarify and structure proofs, and indeed determine the whole shape of a proof, particularly in existence proofs. They are particularly useful when proving large theorems, as the same material does not have to be proved many times in the same proof, and it is easier to keep track of the different levels of proof of the theorem.

Within proof theory, a lemma is a theorem introduced into a proof by use of the cut rule. So, many types of generalisation can be viewed as the use of unproved lemmata. As discussed above, lemmata (and hence the cut rule) may be used both for reasons of construction (to help search for a proof) and presentation (to produce clearer and more efficient proofs). They also allow, via the cut rule, the introduction of new terms and definitions, so that the proof is no longer normal. This is termed a 'proper' use of lemmata, where the non-normal search space will extend all of the normal search space and hence will be larger. However, not all applications of the cut rule introduce new terms: an 'improper' use is when two parts of a normal proof are joined, and in this case the proof will not be more efficient (apart from the possibility of using the lemma as a shorthand in several places), although it might be clearer or more structured.

Various techniques have been proposed to introduce lemmata into theorem-proving: "gazing", used to introduce explicit definitions by abstracting the search space (Plummer & Bundy, 1984); Plaisted's work on abstraction mappings (Plaisted, 1980); and Walsh's work on the implementation of abstraction — see for example (Giunchiglia & Walsh, 1989).

2.3.4 Cut Elimination and Theorem Proving

Cut elimination theorems are important in theorem proving because they enable a restriction of the search space involved when building up a proof. If the cut rule were allowed to be used in the proof, random formulae could be cut in, making automation of the search for a proof practically impossible. On the other hand, if cut elimination were to hold, one would need only consider normal (ie. cut-free)[21] proofs when doing theorem-proving.

This, together with the factors mentioned in Subsection 2.3.2, explains why cut elimination might be desirable. Hence it is unfortunate that when predicate calculus is extended to give a formalisation of arithmetic by the addition of Peano's axioms[22] including induction, cuts may no longer be eliminated in all cases. One such exception is illustrated by the example below — it is not possible to prove $s = t \vdash t = s$ without using cut (although equality could be formulated so that cuts were not needed — in this case by including $s = t \vdash t = s$ as an axiom):

$$\dfrac{\dfrac{\dfrac{}{\vdash s = s}\,axiom}{s = t \vdash s = s}\,lthin \qquad \dfrac{}{s = t, s = s, s = s \vdash t = s}\,basic\ sequent,\ type\ 3}{s = t, s = s \vdash t = s}\,contr$$
$$\dfrac{\qquad\qquad}{s = t \vdash t = s}\,cut$$

Since this is the case, automatic search for a proof might have to involve use of the cut rule, and the cut formula would have to be provided by some sort of heuristic, since there would be no logical rule which would give it. Unfortunately, there is no easy or fail-safe method of generating the required cut formula.

2.3.5 Cut Elimination: Implementation and Results

As mentioned above, cut elimination holds for predicate calculus. An implementation of this process (for simple cases) was carried out within the implemen-

---

[21] A normal proof is one in which every formula is a subformula of the hypotheses or the conclusion.

[22] See Subsection 4.1.1.

tational system Seldon described in Chapter 10[23]. In order to implement the cut elimination theorem, various theoretical considerations had to be made in view of the slightly unusual logical rules (notably "cut") used by the theorem prover selected. Much of the theory given in texts such as (Dummett, 1977) or (J.-Y. Girard & Taylor, 1989) had to be re-worked in order to be implemented. Seldon itself was extended in order to make it more suitable as an implementational system for the proposed research (for example, the logical rule '$\neg$' was added to the system, and a set-theoretic approach was taken in preference to using exchange and contraction rules). The detail is not relevant here[24], but it might be worth mentioning the general algorithm used, which is analogous to those given in order to prove cut elimination in many other systems.

In order to understand the algorithm for cut elimination, there are two main measures of complexity of proofs to be considered: the *rank* of a cut is the sum of the maximum lengths of the left-hand and right-hand paths in the prooftree following application of a cut rule in which the cut formula still remains unaltered (that is, for which the succedent or antecedent, as appropriate, of each sequent associated with a node on that path contains the cut formula). The *degree* of a cut is the number of logical constants in the cut formula. The basic idea of the algorithm is, given a proof of a sequent with cuts, to obtain a proof of the same sequent with cuts of lower complexity, and then repeat to get a cut-free proof. More specifically, a cut with no cuts above it in the tree is selected (where the original sequent is counted as being at the 'bottom' of the tree), and its rank is measured. If the rank is positive, a rank-reduction step is carried out. This involves the replacement of one subtree by another with the same premises and conclusion, but with the difference that any cuts in the second subtree will always be of lower rank than the first. The various cases may be worked out theoretically. The procedure is then repeated. When the rank is zero, the degree of the cut is

---

[23]Further details are given in (Baker, 1990).

[24]The final result is that, upon input of a proof in Seldon, by typing the predicate **eliminate_cut**, a cut-free proof of the same sequent will be returned (Baker, 1990).

measured. If this is zero, the cut may be directly eliminated, otherwise the cut must be replaced by one(s) of lower degree. Unfortunately, this may increase the rank. The procedure is then repeated until all cuts are eliminated: note that since the rank may always be strictly reduced without change of degree, while the degree may always be strictly reduced, and cuts of rank and degree zero may always be eliminated, then it follows that the process will terminate, and that all cuts may be eliminated.

Although cut elimination does not hold for Peano Arithmetic, there are other systems for which it does: for example, there is cut elimination for primitive recursive arithmetic (Kreisel, 1965, P161). The latter is quantifier-free arithmetic encompassing propositional calculus, primitive recursive functions and induction; for further details see (Takeuti, 1987, P84). In addition, Schütte proved that all but the most trivial applications of the cut rule (for which the cut formula is of the form $s = t$) can be eliminated for the system encompassing Peano Arithmetic together with the ω-rule, PA_ω (see Chapter 5) (Schütte, 1960)[25], and this was followed by consideration of other systems for which cut elimination also holds, such as Peano Arithmetic with a recursively restricted ω-rule (Shoenfield, 1959).[26] For further details of cut elimination in the system $L_{\omega_1\omega}$ (cf. Section 2.6), see (Girard, 1987, P393–4).

In order to carry out cut elimination in PA_ω, an assignment of ordinals is provided which not only measures the "lengths" of proofs but also gives direct estimates of number-theoretic bounds for existential theorems. The basic idea underlying cut elimination is analogous to that for predicate calculus given above, and the essential idea is as follows: consider a proof in PA_ω of Γ of the following form, where the cut-formula A is either $\forall x B(x)$ or $B_0 \wedge B_1$:

$$\dfrac{\dfrac{\Gamma \vdash B_i \; all \; i}{\Gamma \vdash A}\forall\omega \; or \; \wedge r \qquad \dfrac{\Gamma, B_n \vdash C \; some \; n}{\Gamma, A \vdash C}\forall l \; or \; \wedge l}{\Gamma \vdash C} \; cut(A)$$

[25]See the Appendix of (Mendelson, 1964) for Schütte's consistency proof.

[26]See also (Girard, 1987, P393–4).

This proof may be replaced by:

$$\frac{\Gamma \vdash B_n \quad \Gamma, B_n \vdash C}{\Gamma \vdash C}\ cut$$

where the cut-formula B_n is a subformula of the original cut-formula A. Repetition of this idea yields a proof of Γ in which all the cut-formulae are prime.[27] For a formal treatment, see (Schwichtenberg, 1977).

2.3.6 Conclusions Regarding Cut Elimination

This section has provided essential background for the overall concern of the thesis with the automatic derivation of proofs within some formalisation of arithmetic. The importance of cut elimination has been discussed, by means of justification of the consideration in Chapter 4 of a formalisation of arithmetic for which cut elimination is valid. Moreover, cut elimination will prove to be highly relevant during the discussions about generalisation which follow in later chapters.

The next sections consider aspects of infinitary proof including Gödel encoding, various ω-rules, infinitary logics and, lastly, motivation for consideration of systems of arithmetic with ω-rules.

2.4 Gödel Encoding

The ω-rule is not suitable for implementation, as it has an infinite number of premises, but various restricted forms might be suitable. This section briefly discusses Gödel numbering, which provides the usual approach to placing restrictions on the ω-rule and describing systems with an infinitary rule.

One of the essential ideas, that of coding numbers, which Gödel used in his proof of the incompleteness theorem has become a standard procedure in logic. Information may be presented in many different symbolic languages, but it is often

[27] A formula is prime if it is of the form $\bot$ or $\beta(t_1, \ldots, t_n)$, where β is a decidable predicate and the t_i are terms.

convenient to put this into numerical form in order to discuss and manipulate this information. For example, information processed by machines is often first converted to a numerical form (although it is also possible to deal directly with objects such as formulae in most modern programming languages). Gödel gave a construction for an arbitrary first order language $\mathcal{L}$ such that each symbol, term, formula and sequence of formulae of $\mathcal{L}$ is assigned a code number in such a way that there is an effective means of going from formal objects to their Gödel numbers, and vice versa. There are different ways of doing this, but the general procedure is to use the expression as a product of powers of primes.

Gödel numbering is a one-to-one map from the formal expressions of the language to the natural numbers. The corresponding number, called the Gödel number, of an expression S is denoted by $\ulcorner S \urcorner$. An operation or relation defined on a class of formal objects is said to be arithmetised if it is thought of in terms of a corresponding arithmetical operation on the Gödel numbers of these objects. The operations of substitution and the generation of the nth numeral may be arithmetised in this way. A proof or deduction will be a finite sequence of strings of symbols, and so will also have a Gödel number.

Gödel's purpose in devising this coding system was to transform assertions about a formal system (such as PA) into assertions about numbers, and then to express these assertions within the formal system. The assertions which can be made about a formal system concern formulae, theorems and proofs. There is a primitive recursive relation $P_{\mathcal{K}}(m, n)$ which holds if and only if m is the Gödel number of a sequence of formulae which constitutes a proof in $\mathcal{K}$ of the formula whose Gödel number is n. $\exists x\ P_{\mathcal{K}}(x, \ulcorner A \urcorner)$ is often abbreviated to $Prf_{\mathcal{K}}(\ulcorner A \urcorner)$ (or $\vdash_{\mathcal{K}} \ulcorner A \urcorner$). Other properties of the system give rise to new relations in a similar way. Now the expression of P in PA provides a formula which would decide within the system itself the meta-level question of whether any sequence of formulae constitutes a proof in PA. This provides an attempt to use PA as a metasystem for itself. However, a contradiction is avoided since this attempt can only be partial because not all relations (over $\mathbb{N}$) are expressible in PA.

Further discussion of restrictions on the ω-rule using an encoding approach is given in Subsection 4.3.2.

2.5 Various Omega Rules

A standard form of the ω-rule is:

$$\frac{A(0), A(\underline{1}) \ldots A(\underline{n}) \ldots}{\forall x A(x)}$$

where $\underline{n}$ is a formal numeral, which for natural number n consists in the n-fold iteration of the successor function applied to zero, and A is formulated within the language of arithmetic. Löpez-Escobar has claimed that the ω-rule is more basic than induction (Löpez-Escobar, 1976). This rule is not derivable in Peano Arithmetic (PA)[28], since for example, for the Gödel formula $G(x)$, for each natural number n, $PA \vdash G(\underline{n})$ but it is not true that $PA \vdash \forall x G(x)$. As mentioned above, cut elimination may be carried out on the resulting system, which makes it easier to prove consistency.

However, this is not a good candidate for implementation since there are an infinite number of premises. It would be desirable to restrict the ω-rule so that the infinite proofs considered possess some important properties of finite proofs. There are many such restrictions, which shall be considered below. One suitable option which has already been mentioned is the **constructive ω-rule**. The ω-rule is said to be constructive if there is a recursive function f such that for every n, $f(n)$ is a Gödel number of $P(n)$, where $P(n)$ is defined for every natural number n and is a proof of $A(\underline{n})$ (Takeuti, 1987). This is equivalent to the requirement that there is a uniform, computable procedure describing $P(n)$, or alternatively that the proofs are recursive (in the sense that both the prooftree and the function describing the

[28] A formalisation is given in Section 4.1.

use of the different rules must be recursive) (Yoccoz, 1989a), which is the basis taken for implementation (as opposed to the Gödel numbering approach).[29]

The sequent calculus enriched with the constructive ω-rule in place of the rule of induction (let us call it $PA_{r\omega}$[30]) has cut elimination, and is complete (Shoenfield, 1959). Moreover, $PA_{r\omega} + IND$ is a conservative extension of $PA_{r\omega}$.[31] The principle of induction is a consequence of the ω-rule (see Section 7.2). However, the compactness theorem fails, as does the second-order interpolation theorem.

Shoenfield has shown that 'PA + ω-rule' (PA_ω)[32] is a conservative extension of 'PA + recursively restricted ω-rule' (Shoenfield, 1959) and this result has been extended to weak second order arithmetic by Takahashi (Takahashi, 1970). However, this does not apply to all theories, and notably not to some cut-free systems, nor to HA, intuitionistic number theory (López-Escobar, 1977). The usual formalisations of intuitionistic arithmetic produce a stronger theory with the unrestricted ω-rule than with the recursively restricted ω-rule; there is a similar result for some cut-free systems, and analogously for the recursively versus primitive recursively restricted ω-rule. So, intuitionistic primitive recursive trees are not enough to represent $HA_{r\omega}$ (and therefore not enough to enable completeness of $HA_{r\omega}$). However, if HA^* is defined as HA together with the transfinite induction schema over all well-founded[33] primitive recursive (binary) relations on natural numbers, then HA^* is complete for Π^0_2 sentences. Furthermore, in any metatheory extending second order arithmetic, this theory is equivalent to HA with the full ω-rule (Friedman & Scedrov, 1985). To show this, recursive well-founded prooftrees are

---

[29]Indeed, the introduction of infinitary rules in most proof systems in computer science can be restricted to a recursive one (Yoccoz, 1989b).

[30]For a more formal description see Chapter 5.

[31]A is a *conservative extension* of B iff, for each closed formula F of the language of the theories, $B \vdash F \leftrightarrow A \vdash F$.

[32]See Section 4.2 for description.

[33]A binary relation R on a set S is said to be well-founded if
$\forall X\ (\forall a \in S(\forall b \in S(bRa \rightarrow b \in X) \rightarrow a \in X) \rightarrow \forall a \in S\ a \in X)$.

considered and coding is used: for the infinitary rules one requires a recursive sequence of Gödel numbers of trees that end with the premises; see Subsection 4.3.2 for further details of such a method.

Given the system of intuitionistic arithmetic HA defined by Kleene (Kleene, 1962), a proof that HA_ω (the system HA with the ω-rule replacing the rule of induction) is stronger than $HA_{r\omega}$ (the analogous system with a recursive restriction on the ω-rule), runs as follows. Assuming the model $\mathcal{M} =< \omega, +, ., ', \int >$, it is provable by induction that for all sentences A of HA, if A is true in $\mathcal{M}$, then $HA_\omega \vdash A$. In addition, if $HA_{r\omega} \vdash A$, then A is realisable.[34] However, $\forall x(\exists y T(x,x,y) \vee \forall y \neg(T(x,x,y)))$ is true but not realisable (López-Escobar, 1977, P92). Hence, HA_ω cannot be a conservative extension of $HA_{r\omega}$.

Another rule which has been proposed is the finitely-applied ω-rule:

$$\frac{\vdash \forall x \exists y P_{PA}(y, subst(\ulcorner A(z) \urcorner, x))}{\vdash \forall x A(x)}$$

where $subst(\ulcorner A \urcorner, n)$ is the Gödel number of the sentence obtained by substituting $\underline{n}$, the numeral representing the number n, for x in $A(x)$. This ω-rule is a finite formulation of the ω-rule by means of arithmetisation of syntax, and is hence intended to replace the latter. There is a strong correspondence between this rule and the uniform reflection principle (given in Section 11.1). Note that in particular the previous rules considered differ from this syntactically finite form of the ω-rule which was considered by Rosser (Rosser, 1937) and subsequently Feferman, for which completeness results rely upon the particular path taken which indexes iteration through the constructive ordinals. More specifically, Rosser showed that Gödel's incompleteness theorem applies for the resulting system of arithmetic with this form of the ω-rule, $\mathcal{Z}'$, and also obtained incompleteness for arbitrary finite iterations of the rule; (Feferman, 1962) and (Feferman & Spector, 1962) give completeness and incompleteness results respectively, depending on the path taken for the iteration into the recursive transfinite of the procedure by which the system $\mathcal{Z}'$ can be obtained from $\mathcal{Z}$, the analogous system without this rule. Feferman has

[34] For a discussion of realisability, see (Dummett, 1977, P318).

also shown the equivalence of extending PA by the finitely applied ω-rule with extending PA by uniform reflection[35] (Feferman, 1962); it is also equivalent to an extension by ϵ_0-transfinite induction, as mentioned in Section 11.1. Restriction of this rule to primitive recursive formulae is weaker than the ordinary ω-rule restricted to primitive recursive formulae, but still strong enough to generate a proper extension to PA, since for example the Gödel sentence may be derived.[36] Another form of the rule, the finitely applied ω-rule as an extension of primitive recursive arithmetic[37], considered for example by Ignjatovic (Ignjatovic, 1988), extends primitive recursive arithmetic, but fails to extend PA.

Hence, there are many versions of a restricted ω-rule. The recursively restricted one is considered in particular because the resulting system is complete and it is suitable for automation. There is a primitive recursive counterpart[38] which would also be suitable, for which the resulting system was shown to be complete by Nelson (Nelson, 1971). This is in fact the ω-rule chosen for the basis of implementation; see Subsection 4.3.1 regarding the decision to use the primitive recursive rather than the recursively restricted ω-rule. Since the primitive recursive version (cf. Chapter 5) is also a constructive ω-rule, to avoid confusion I shall refer to the "recursively restricted ω-rule" and the "primitive recursively restricted ω-rule", whenever differentiation is applicable.

Next I consider briefly another way in which infinite proofs may be represented.

---

[35]See Section 11.1.

[36]Any sentence A such that $A \equiv \neg \vdash A$ is provable in a system S is called a Gödel sentence for S (Takeuti, 1987, P79).

[37]See (Girard, 1987, P67).

[38]In other words, such that there is a primitive recursive function f for which, for every n, $f(n)$ is a Gödel number of the proof of $A(\underline{n})$, the nth numerator of the ω-rule.

2.6 Infinitary Logics

Infinitary logic absorbs the notion of infinity into its syntax by allowing infinite disjunctions and conjunctions. If α and β are two infinite cardinals, then the infinitary logic $L_{\alpha\beta}$ is first order logic, but allowing the conjunction and disjunction of a set of fewer than α formulae and universal and existential quantification over a set of fewer than β variables, such that there are α individual variables. Hence, $L_{\omega\omega}$ is classical first order logic. The next weakest case is the language $L_{\omega_1\omega}$, which allows countable conjunctions and disjunctions but only finite strings of quantifiers. That is to say that given a first order language L with countably many relation, function, and constant symbols, and ω_1 variables, the language $L_{\omega_1\omega}$ has the same symbols as L, but in addition the conjunction symbol and disjunction symbol may be applied to countable or finite sets of formulae as well as to pairs of formulae. From around 1962, the model theory of $L_{\omega_1\omega}$ has been investigated; the Lowenheim-Skolem Theorem holds for $L_{\omega_1\omega}$ but not the compactness theorem, and neither can the concept of a well-ordering be expressed. Well-behaved sublanguages are often considered instead, since otherwise there are uncountably many formulae. $L_{\omega_1\omega_1}$ is a much more powerful logic than $L_{\omega_1\omega}$; see (Keisler, 1971) or (Takeuti, 1987, P188–284) for further details of infinitary logics. Even the simplest infinitary logic, $L_{\omega_1\omega}$, is not suitable for implementation because it would not be possible to represent infinite conjunctions of arbitrary formulae in a finite way (although of course there is the possibility of allowing just effectively presented infinite conjunctions, but these would only form a subset of such conjunctions of arbitrary formulae).

2.7 Motivation

The ω-rule, and also its constructive counterparts, are of interest for several reasons. First, they may be seen as an attempt to overcome the limitations of reasoning using a formal system that follow from Gödel's first incompleteness theorem. As mentioned earlier in this chapter, Gödel showed that there are statements in the formal language of natural number arithmetic that are logical consequences of the usual axioms of arithmetic but cannot be proved in the first-order theory, unless that theory is inconsistent (Gödel, 1931). His procedure was to construct in a formal system of simple type theory with the natural numbers as ground type, $\mathcal{P}$, a formula A which states that there is no natural number x which is the Gödel number of a proof in $\mathcal{P}$ of the formula B, with B set to be A itself. In other words, A may be defined as $\neg\exists x(\Pi_x \; proves \; A)$, where Π_n is the proof (with Gödel number n) of some theorem in $\mathcal{P}$. A is essentially stating "I am unprovable (in $\mathcal{P}$)" (which is analogous to the liar paradox, with "unprovable" instead of "false"). Now, if $\mathcal{P}$ has the feature that only true formulae are provable (which is desirable!), then with strong similarity to both the liar paradox and Russell's paradox, if the Gödel formula A were provable (in $\mathcal{P}$), it must be false, and hence unprovable (giving a contradiction). Similarly, $\neg A$ is also unprovable. Therefore, A is formally undecidable in $\mathcal{P}$. Note that a first order theory is complete if and only if for each closed formula of the language of the theory either A or $\neg A$ is provable, and consistent if and only if at most one of A and $\neg A$ is provable. Informally, a complete theory is one strong enough to allow proof of any statement that it would be desirable to prove, and a consistent theory is one free of formal contradiction, so the incompleteness theorem says that for any formal theory of arithmetic there will always be true statements that are not theorems of this theory, and hence we can never completely formalise arithmetic.[39] As discussed in

[39]However, according to D. Isaacson, it is possible to define a concept of arithmetical truth with respect to which PA is complete, as in (Isaacson, 1987).

Subsection 2.1.1, Penrose argues that arithmetic is non-algorithmic in nature, and that mathematical insight allows us to see the truth of true but unprovable statements, and J.R. Lucas similarly uses Gödel's proof to draw a distinction between men and machines (Lucas, 1970).

In particular, Penrose says that the ω-rule is non-algorithmic, in the sense of being computationally infeasible because it requires the proof of an infinite number of hypotheses, and Gödel's theorem shows that the usual computable approximation to the ω-rule, namely induction (or even a finite collection of new induction rules), is an inadequate substitute for the ω-rule. From an analysis of Gödel's incompleteness theorem it can be seen that the problem lies in the formalisation of mathematical induction, but on the other hand Gödel's theorem does not apply if the ω-rule is used since this is not a formal system in the required sense: it is rather a semi-formal system because the proofs are infinite. Therefore, use of the complete formalisation of arithmetic constructed from the Peano axioms and the constructive ω-rule would provide an objection to this argument of Penrose's, since a complete formalisation of arithmetic can be constructed from the Peano axioms and the constructive ω-rule (Shoenfield, 1959) and since this system may be implemented (and is therefore algorithmic in nature).

From an implementational point of view, and quite apart from these considerations, completeness is desirable because it would be highly undesirable that a formula of the system should be seen by elementary computation to be false and yet be provable. Although resource limitations, such as time, may not allow the full exploitation of completeness, at least by having a complete system the situation where simple or interesting deductions are just not possible is avoided.

A related reason for investigating use of the ω-rule in theorem-proving is that it is possible to prove more expressions in a system including it, for "PA + constructive ω-rule" is a stronger system than PA. I shall be especially concerned in Section 7.1 with certain expressions which can be proved in a system with the constructive ω-rule but which may not be provable in Peano Arithmetic without cut.

Secondly, the ω-rule allows us to do away with the cut rule. Many meta-theorems follow from the cut elimination theorem in both first and higher-order proof the-

ory. However, as discussed in Section 2.3, it is impossible to eliminate all cuts in extensions of arithmetic because the formal proofs contain applications of mathematical induction. Yet, by means of introducing the ω-rule, Schütte eliminated all applications of induction and all non-trivial applications of the cut rule in first order arithmetic (Schütte, 1960).[40] Hence, an important technique for investigating derivability in formal systems of arithmetic has been to embed such systems into semi-formal systems with the ω-rule (although there have been other motivations for this, such as searching for consistency results).[41]

In addition, some proofs may be generated more easily in the system with the ω-rule, and there is scope for a new approach to the problems of generalisation, as shown in Chapter 8.

The following subsection presents further motivation for consideration of the ω-rule in terms of its practical use in automated-theorem proving.

2.7.1 Use of the Constructive Omega Rule

One use of the constructive ω-rule is to enable automated proof of formulae, such as $\forall x\ (x+x)+x = x+(x+x)$, which cannot be proved in the normal axiomatisation of arithmetic without recourse to the cut rule (notably, induction does not carry through). As mentioned above, in these cases the correct proof could be extremely difficult to find automatically. However, it is possible to prove this equation using the ω-rule by generating proofs of $(0+0)+0 = 0+(0+0)$, $(1+1)+1 = 1+(1+1), \ldots$ and working out the general pattern of these proofs. In Chapter 7 another reason why it might be advantageous to carry out theorem-proving in the system $PA_{c\omega}$

---

[40]Parsons went on to prove that every proof with finite order (in the sense of having a uniform bound on the number of constructional inferences in each branch) in Schütte's system $\mathcal{S}$ of number theory with the ω-rule, can be effectively transformed into a proof from certain free variable axioms in Schütte's quantification theory K_1 (Parsons, 1958).

[41]Many results relevant to the understanding of nonconstructive mathematics from a constructivist point of view can be obtained from consistency proofs; for example, in 1956 Beth proved classically the completeness of intuitionist quantification theory.

is presented, namely that induction may carry through when it does not in the system PA. Next, a preview of the general implementational approach which will be taken will be given.

For the implementation it is necessary to provide (for the nth case) a description for the general proof in a constructive way (in this case a recursive way), which captures the notion that each $P(n)$ is being proved in a uniform way (from parameter n). This is done by manipulating $A(\underline{n})$, where $\forall x A(x)$ is the sequent to be proved, and using recursively defined function definitions of PA as rewrite rules, with the aim of reducing both sides of the equation to the same formula. The recursive function sought is described by the sequence of rule applications, parametrised over n. In practice, the first few proofs will be special cases, and it is rather the correspondence between the proofs of $P(99)$, say, and $P(100)$, which should be captured. The processes of generation of a (recursive) general proof from individual proof instances, and the (metalevel) checking that this is indeed the correct proof have been automated (see Chapter 10). Further details of the algorithms and representations used will be given in Chapter 10, together with in Chapter 6 the correspondence between the adopted implementational approach and the formal theory of the system described in Subsection 4.3.2.

2.7.2 Conclusions Regarding Motivation

In this section interest in ω-rules within automated deduction has been justified, and a suitable rule for my purposes chosen, namely a constructive version of the ω-rule, which shall henceforth be used as the basis for implementation. Also, an indication of how the ω-rule may be used in practice has been given.

2.8 Summary of Chapter

In this chapter various ω-rules have been introduced, and a coherent view of infinite proofs has been presented, with reference to differing viewpoints within the foundations of mathematics, in order to assess the nature of the various types of ω-rules and to provide a notion of soundness for the system $PA_{c\omega}$ which shall be considered in Chapter 5. The ω-rule was seen to provide a canonical proof, and hence a meaning, for universally quantified statements, and implementation of such an infinitary rule may be carried out in a constructive manner.

In Section 2.3 cut elimination was defined and discussed. The cut elimination theorem for predicate calculus states that every proof may be replaced by one which does not involve use of the cut rule. This theorem no longer holds when the system is extended with Peano's axioms to give a formalisation for arithmetic. The problem of generalisation results, since arbitrary formulae can be cut in. This makes theorem-proving very difficult — one solution is to embed arithmetic in a stronger system, where cut elimination holds. In particular, this is the case given the addition of the constructive ω-rule. In this new system, automated proof of any arithmetic sequent could be attempted, without hindrance from the problems of generalisation.

In Section 2.7 various motivations for consideration of the system of arithmetic with a primitive recursively restricted ω-rule in place of induction ($PA_{c\omega}$) were presented. Some advantages of the approach are

- the completeness of the resulting system; by this means it is possible to overcome limitations of reasoning using a formal system that follow from Gödel's first incompleteness theorem (namely that there are true statements that are not theorems of the formal system). Hence, more expressions are provable in $PA_{c\omega}$ than PA (and especially than $PA \setminus cut$).

- no generalisation is needed in this system since cut elimination for $PA_{c\omega}$ holds. Hence desirable properties such as the subformula property may be derived.

- some proofs are generated more easily in $PA_{c\omega}$ than PA.

- as shown in Chapters 8 and 9, there is a method by which proofs in $PA_{c\omega}$ may guide proofs in PA.

However, the explicit manipulation of proofs necessitated by use of the ω-rule has both advantages and disadvantages, as discussed in Section 11.1.

Chapter 3

Context

"But how can finite grasp infinity?"

John Dryden

This chapter presents related work which will be built upon in later chapters. In Section 3.1, an overview of generalisation is given: an analogy of generalisation for natural deduction systems is given in Subsection 3.1.1, followed by classification of different types of generalisation. In Section 3.2, an overview of inductive theorem proving techniques during the last few decades is given, since the implementation presented in this thesis is closely related to this area of research. Section 3.3 deals with inductive inference, which is a notion encompassing some of the algorithms used in further chapters. Since the technique of explanation-based generalisation is used later in this thesis (in particular Subsection 8.1.4 relates to its application to a new domain), it is described in Section 3.4. In addition, in Section 3.5 a description is given of primitive recursion over abstract datatypes.

3.1 Generalisation

A general overview of generalisation within automated deduction has already been provided in Section 1.3. In this section an overview of generalisation as a process of enabling proof of statements by induction will be given, together with an analogy for natural deduction and the presentation of various categories of generalisation methods.

If induction is blocked for an expression[1], generalisation may be used as a step to convert this expression into a new, more general, expression which may be proved by induction. A generalisation of a statement Φ is a statement Ψ such that $\models (\Psi \rightarrow \Phi)$. Therefore, if verification of Φ is the goal, it is sound to prove Ψ instead of Φ. Indeed, it might actually be *necessary* to adopt this approach in cases where Φ is a theorem of a system $\mathcal{S}$ but it is not the case that Φ may be proved by induction within $\mathcal{S}$ (although Ψ may be found such that Ψ is provable by induction within $\mathcal{S}$, and such that Ψ is a generalisation of Φ).

Heuristics are needed for the suggestion of Ψ since there is no appropriate algorithm for finding suitable generalisations.[2] A suitable suggested generalisation will be constrained by being just general enough to provide a proof by induction whilst avoiding overgeneralisation, for not only may generalisations Ψ be computed such that it is not the case that Ψ is provable by induction, but overgeneralisations may also be computed (that is to say where Ψ is not a theorem of $\mathcal{S}$).

---

[1]Induction is said to blocked if, after all available symbolic evaluation has been carried out, the induction conclusion is still not an instance of the induction hypothesis, and hence remains unprovable.

[2]Of course, an algorithm could be used which tried every possible option, but the search required might be infeasibly large, plus there would be the drawback of not allowing the conclusion that there was no adequate generalisation. In short, this trivial approach is not what is required.

3.1.1 Analogy of Generalisation in Natural Deduction Systems

In sequent calculus, generalisation is provided by the cut rule. This has an analogy in the formalisation of other systems. In particular, normalisation for natural deduction systems corresponds to cut elimination in sequent calculus.[3] Hence, doing a cut corresponds to having a non-normal proof. A normal deduction proof will have a particular form, for each branch in the deduction may be divided in the following way: there will be a formula A such that all formulae above A are premises of applications of elimination rules and all formulae below A (with the exception of the last) are premises of applications of introduction rules.

Normalisation may always be achieved in classical and intuitionistic logic (corresponding to the property of cut elimination in predicate calculus). Arithmetic can be represented in natural deduction by allowing the axioms of arithmetic to appear as the leaves of the proof (ie. the hypotheses), and not worrying if they do not get discharged; such a formulation is given in (Troelestra, 1982, P19–21), in which the induction rule is naturally a rule of inference rather than an axiom.

Generalisation does not occur in the usual natural deduction system, but on the other hand one may achieve this effect. It is possible to look upon a proof in a calculus of sequents as an instruction on how to construct a corresponding natural deduction (see (Prawitz, 1965, P91)). In general, the rules on the succedents correspond to the introduction rules whereas the rules on the antecedents correspond to the elimination rules. For example:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \longleftrightarrow \frac{A \quad B}{A \& B} \& I$$

In particular,

$$\frac{\Phi(t), \Gamma \vdash \Delta}{\forall x \Phi(x), \Gamma \vdash \Delta} \longleftrightarrow \frac{\forall x A}{A_t^x} \forall E$$

---

[3] See (Prawitz, 1965), and the discussion in Subsection 2.1.1.

The generalisation step from $\forall z \Phi(z, z)$ to $\Phi(x, y)$ corresponds to:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Phi(x,y)}{\forall y \Phi(x,y)}\forall I}{\forall x \forall y \Phi(x,y)}\forall I}{\forall y \Phi(z,y)}\forall E}{\Phi(z,z)}\forall E}{\forall z \Phi(z,z)}\forall I$$

These steps achieve the same as a generalisation step, and furthermore they form a non-normal proof fragment (and necessarily so in order to perform this transformation). There will always be a proof fragment of this form when making such a 'generalisation step', and it will vary only according to the number of variables involved.

The overall conclusion to be made is that the same search problem occurs in natural deduction systems as in other formulations, such as the sequent calculus used for implementation.

3.1.2 Different Types of Generalisation

In this subsection terminology related to different kinds of generalisation shall be presented, including the broad approach of considering generalisation in terms of inversion of sound inference rules. A well-known alternative method of classification of some of the main types of generalisation used in theorem-proving will also be given: Boyer and Moore are normally associated with generalising terms to variables, and Aubin with extending their work to allow generalisation of variables apart and generalisation of terms with a constant in an accumulator position; for further details of such methods, plus further developments for example by Hesketh, see Section 11.3.

Generalisations may be obtained by backwards application of sound (derived) first-order inference rules. Examples of such rules are given in Table 3–1. The substitution rule computes a generalisation of Ψ by replacing one or several occurrences of

a term in Ψ by a universally quantified variable. An example of its use for generalisation is that the rule of substitution applied backwards to $\forall x\ fac(x).0 = 0$ would yield a formula $\forall y\ y.0 = 0$. The inversion of the weakening rule is mainly applied to evaluated step formulae for elimination of induction hypotheses, and inverted replacement corresponds to Aubin's notion of indirect generalisation[4] (Aubin, 1976). Inverted modus ponens is used to obtain simpler verification problems and to avoid overgeneralisations computed by inverted substitution, and is used by Boyer and Moore (via labelling of some lemmata as generalisation lemmata by the user, in order for their system NQTHM to consider just those lemmata when computing generalisations) (Boyer & Moore, 1979). With each rule some heuristic must be provided which tries to decrease the risk of an overgeneralisation by stipulating the cases in which the inverted rule should be applied.[5]

substitution	$\frac{\forall\ldots\Phi(\ldots x\ldots)}{\forall\ldots\Phi(\ldots t\ldots)}$
functionality	$\frac{\forall\ldots t_1{=}s_1\wedge\ldots\wedge t_n{=}s_n}{\forall\ldots f(t_1\ldots t_n){=}f(s_1\ldots s_n)}$
splitting	$\frac{\forall\ldots\Phi\wedge\Psi}{\forall\ldots\Phi}$
weakening	$\frac{\forall\ldots\Psi}{\forall\ldots\Phi\rightarrow\Psi}$
modus ponens	$\frac{\forall\ldots\Psi\wedge(\Psi\rightarrow\Phi)}{\forall\ldots\Phi}$
replacement	$\frac{\forall\ldots t{=}s\wedge\Phi[t]}{\forall\ldots\Phi[s]}$

Table 3–1: Useful Inference Rules for Computing Generalisations

However, it might be more practical to consider the following four categories, rather than the classification given in Table 3–1.

Terms to Variables

A generalisation heuristic which works in many cases is to examine any common term structure, and generalise common subexpressions to a new variable (cf. sub-

[4]See Subsection 11.3.2.

[5]See (Walther, 1992) for further details.

stitution backwards). For example (where <> denotes 'append'), naïve induction fails for $\forall x \forall y \forall z (rev(x) <> (y <> z) = (rev(x) <> y) <> z)$, but this may be generalised to $\forall v \forall y \forall z\ v <> (y <> z) = (v <> y) <> z$, for which induction works (see Subsection 8.2.2 for further details).

Variables Apart

If generalisation of terms to variables fails, or is not appropriate, other methods of generalisation may be tried. The idea of generalisation of variables apart is, given some theorem with the same variable occurring many times, to find a generalisation by careful differentiation between the variables. For example, one may generalise $\forall x\ (x+x)+x = x+(x+x)$ (which may not be proved by induction) to $\forall x \forall y \forall z\ (x+y)+z = x+(y+z)$, which may be proved by induction on x (see Section 8.1).

Terms with a Constant in an Accumulator Position

Within the recursive definition of a function, an accumulator is an argument used to accumulate the value of the function, such as the second argument in the definition of $rev2$, where:
$rev2(nil, l) = l$;
$rev2(h :: t, l) = rev2(t, h :: l)$.
A new type of generalisation is needed in these cases. As an example, $rev2(x, nil) = rev(x)$ may be generalised to $rev2(x, a) = rev(x) <> a$. See Appendix C.3 for further details of this example.

Addition of Accumulators

Appendix C also considers examples for which the introduction of an accumulator is needed for generalisation, but not necessarily in order to generalise a constant: for example, $\forall l\ rotate(len(l), l) = l$ may be generalised to $\forall la\ rotate(len(l), l <> a) = a <> l$.

A brief introduction to generalisation has been given in this section, in preparation for further discussion of this topic and the presentation of a new generalisation method in Chapter 8.

3.2 Inductive Theorem Proving

The application area of this thesis lies within the field of inductive theorem proving, and as such, is a development of some of this work. This section will present a brief overview of the various systems developed within this domain; these are especially of interest because all of these systems are faced with the problems of generalisation, since inductive proof is being carried out. Hence, the generalisation technique presented in Chapter 8 could be implemented as a useful addition to any of these systems.

Much work has been carried out in the area of inductive theorem proving over the last few decades. There have been many different approaches, which have involved a varying amount of automation. I shall consider only the major inductive theorem provers, rather than proof environments such as HOL (Gordon, 1988) and Isabelle (Paulson, 1986) (higher order systems in which a variety of object-level logics may be used); the generic environment ICLE (Dawson, 1992) or the Logical Frameworks project. The latter is designed to support proof development in a variety of logics, with main implementation LEGO, which is an interactive proof development system in a natural deduction style, allowing proof development in the logic of calculus of constructions (Luo & Pollack, 1992). The main concern of these proof environments is to enable the user to carry out proofs, rather than to provide methods of automation.

3.2.1 The Boyer-Moore Theorem Prover

Of the inductive theorem provers, perhaps the most famous is that developed since 1972 by Boyer and Moore at Austin, called NQTHM (Boyer & Moore, 1988).

This is able to prove an extensive range of theorems requiring induction in such domains as number theory and LISP. NQTHM is designed to be used mainly with the fixed set of axioms provided, plus a number of definitions given by the user. The Boyer-Moore system incorporates powerful techniques including induction, linear resolution and generalisation. The induction heuristic is closely tied to the principle of recursive definition, and it filters out inductions which are not likely to work and merges competing alternatives. It is semi-automated, in the sense that a user may interact by providing lemmata which might guide the basic prover, support successful statement generalisation, or give hints to recognise termination of an algorithm. It operates on a first order logical system of recursive functions over finitely generated objects, and can express basic datatypes of LISP (in other words, numbers, lists etc.), plus definitions, and recognises well-founded induction schemes.

Theorems which have been proved are added to the rewrite rule block, and labelled according to their specific category. "Methods" (ie. tactics) are tried on the current proof goal sequentially. When one succeeds, the process is repeated on the resulting goal(s). This list of methods will try first symbolic evaluation and various simplifications (such as rewriting, linear and binary resolution and subsumption), and only finally techniques which will complicate the proof (and therefore must be used according to stringent conditions), such as induction and generalisation. It is a heuristic theorem-prover, and so it is possible that no proof may be found, or that it may go down blind alleys. Indeed, the lack of explicit strategy makes it difficult to detect if the proof is in a loop, or if there is generalisation to a non-theorem, and so on.

The prover does much tedious work filling in minor details, but needs the user to input the main steps to the proof for any theorem of interest.

> "Although NQTHM is quite capable of finding proofs for some simple theorems with which even graduate students may struggle, we think of NQTHM as more of a proof checker than as a theorem-prover" (Boyer & Moore, 1990).

Perhaps it is fairer to conclude that it is a very powerful theorem-prover, which is

capable of proving large numbers of theorems automatically, whilst also having an interactive mode. It is able to automatically prove some complicated verification theorems, but the axioms and proofs are implicitly universally quantified: no synthesis proofs are possible as there is no mechanism for existential quantification. However, by using recursive functions, Boyer and Moore are able to express many things usually expressed with quantifiers when dealing with "finite" objects such as trees of integers. Aubin developed his work in inductive theorem proving at Edinburgh (Aubin, 1975); he considered the computation of induction axioms and proposed several techniques for statement generalisation (see Subsection 11.3.2).

3.2.2 OYSTER-CLAM

The OYSTER-CLAM system developed by Bundy and colleagues at Edinburgh (Bundy *et al*, 1990) provides a degree of automation of proofs. The OYSTER system is an interactive proof editor which is the reimplementation in Prolog of the NuPRL program development system[6] by Christian Horn, a visitor to the Mathematical Reasoning Group in Edinburgh (Horn & Smaill, 1990). Many details about the system are already present in NuPRL: the object level-logic is a higher-order constructive logic including induction, in a sequent-calculus formation. Since the object-level logic is constructive, extract terms may be obtained from proofs, and then executed by application to an input. A theorem may be regarded as a specification which is realised by its extract term, and so the system allows program synthesis (see Subsection 10.2.1 for further details about extract

[6]NuPRL is a proof development environment (which builds upon LCF) developed by Constable (Constable *et al*, 1986) for a version of Intuitionistic Type Theory based on that developed by Per Martin-Löf, (Martin-Lof, 1970). (LCF is a proof assistant which stands for 'Logic for Computable Functions', and is an interactive system based on classical logic, in which tactics may be used, in some cases to handle subgoals automatically (Gordon *et al*, 1979).) NuPRL uses constructive logic and embodies a "propositions as types" and "proofs as programs" approach. Like NQTHM, this is more an interactive system for proof development than an automatic theorem-prover. There is a tactic language, but there is no meta-level planning or exploration of the prooftree; nodes of a prooftree are operated upon sequentially and all arguments to inference rules must be instantiated.

terms.) Both definitions and libraries of theorems may be used. Proofs are constructed in a top-down way via application of rules of inference, guided either by a user or by a proof tactic. The latter is a Prolog program which incorporates commands of the theorem proving system (in themselves Prolog predicates). The tactic may, for example, recognise if the sequent to be operated upon corresponds to a certain pattern, and in this case perform certain rules of inference upon it.

CLAM is a meta-level system built on top of OYSTER to turn the latter into an automatic theorem proving system, and support the use of proof plans (van Harmelen, 1989). For each tactic written in the OYSTER system, CLAM contains a specification of the tactic, called a method, which consists of an input formula, preconditions, output formulae and postconditions. Thus CLAM is able to predict if a particular tactic will be applicable in OYSTER without actually running the tactic. A proof plan is obtained via CLAM by finding an applicable method, calculating the output formulae and postconditions, and repeating the process until no further unproved formulae remain. Since, for a given sequent, more than one method may be available, backtracking may be necessary. The whole meta-level process is called proof planning. The proof plan, or record of methods, can then be executed at the object-level by using the corresponding OYSTER tactics. The methods, tactics and heuristics of CLAM are modelled on the Boyer-Moore theorem prover.

3.2.3 INKA

The INKA system (Biundo *et al*, 1986) developed in Karlsruhe and then at Darmstadt and Saarbrücken by Walther and co-workers also has a high degree of automation. It is able to automatically guide symbolic evaluation and lemma recognition when proving the induction step (Hutter, 1990). Hutter compares the induction hypothesis and conclusion to obtain a syntactical pattern which must hold for each intermediate stage of the deduction. INKA also automates termination proofs for recursively defined algorithms (Walther, 1988). Many induction lemmata used by Boyer and Moore may be completed by the system without user

guidance, and statements containing existential quantifiers may also be dealt with. In addition, several generalisation techniques have been implemented within the system: Hummel has described and analysed various generalisation heuristics for recursive functions, some of which she then implemented within the INKA system (Hummel, 1987).

3.2.4 Inductionless Induction

Another, alternative approach to automated induction is that of inductionless induction (so-called because an induction proof is not explicitly constructed), also known as proof by consistency, and inductive completion. These techniques have been developed over the last decade, for example by Huet and Hullot (Huet & Hullot, 1982). Given the set of all statements $\mathcal{T}$ provable from the database of a theorem-prover, the type of approach taken is that if $\mathcal{T}$ is complete, then verifying the consistency of $\mathcal{T} \cup \{\Phi\}$ for some statement Φ is equivalent to showing that $\Phi \in \mathcal{T}$, in other words that Φ is provable in the system. More specifically, if the input language is restricted to universal quantification, the Knuth-Bendix procedure can be used to verify consistency. This procedure is to generate a canonical rewrite system corresponding to $\mathcal{T} \cup \{\Phi\}$ that is consistent precisely when Φ is an inductive consequence of $\mathcal{T}$, and inconsistent when it is not. So, if the process of generating a complete system terminates and a contradiction results, it is concluded that Φ cannot be proved; otherwise $\Phi \in \mathcal{T}$ is proved. The properties of $\mathcal{T}$ and the proof procedure used determine the particular test for consistency. The major calculation is of superpositions of terms, which yield critical pairs; the generation of these pairs corresponds to induction schemas, their simplification corresponds to post-induction proof, and generalised statements and lemmata may also be computed from them (see (Thomas & Watson, 1991), and the following section). Although there is no explicit induction used, Reddy has shown that many inductive completion axioms can be viewed in terms of performing induction over a Noetherian ordering (see Subsection 4.1.2) given by $\mathcal{T}$, in the sense of being a uniformly terminating term-rewriting system where no term can be infinitely rewritten (Reddy, 1990). Reddy used a method called term-rewriting induction to

prove properties of term-rewriting systems, and Knuth-Bendix completion-based inductive proof procedures to construct term-rewriting induction proofs. Goguen has shown the correctness of algebraic methods for deciding the equivalence of expressions by applying rewrite rules, and proving equational inductive hypotheses without using induction (Goguen, 1980). See (Barnett *et al*, 1992) for a comparison of inductive completion with recursion analysis and the rippling technique of CLAM (although this deals with non-definitional lemmata, and not generalisation).

3.2.5 The Chosen Approach

I have not used an inductive completion-type approach, but have preferred the Boyer-Moore-type approach, primarily because the input language is not restricted to universal quantification, and although the system PA, with the ω-rule in place of induction (see Chapters 4 and 5), does happen to be complete, greater flexibility in the general approach would be desirable. Note that in practice anyway it is not claimed that the implementational system used is complete (Chapter 6). The systems which have most influenced my work are those of Boyer and Moore, Aubin and OYSTER-CLAM. The implementation is actually written as an extension of OYSTER (although a different logic is used).

3.3 Inductive Learning

This section refers to related work in the field of inductive inference (in other words, presenting algorithms to obtain a general pattern from individual instances), for part of the work encompassed by this thesis is to implement use of the constructive ω-rule. This implementation may involve the automatic generation of general proofs from individual cases, which corresponds to the extraction of a pattern from a sequence of proofs. This is not a new problem: the first results on automated induction first began to emerge in the mid-sixties and seventies.[7]

Inductive (or experimental) learning seeks by examining examples and non-examples of a concept to determine by syntactic means alone which features in the examples led them to be classified in this way; thus inductive generalisation may be thought of as the process of hypothesising a general rule from examples. Explanation-based (also called analytic) learning also takes into account the process which caused this classification. Analytic techniques permit learning from a single example, whereas inductive learning usually requires many examples to learn a concept. My work has parallels with the work done by Mitchell, and also Shavlik, in the field of (analytic) learning by experimentation (Mitchell *et al*, 1981; Shavlik & Jong, 1989). Shavlik and De Jong have developed a model (namely Physics 101) for applying explanation-based learning to mathematically-based domains. The solution to a specific problem is generalised into a form that can later be used to solve conceptually similar problems. It differs from other explanation-based approaches to learning in mathematical domains, for example by O'Rorke (O'Rorke, 1987), in that it tackles the generalisation of number. However, a greater parallel is that with work in the field of inductive generalisation, rather than such analytic generalisation. Inductive inference, or generalisation, encompasses for instance the early

[7]For example, Winston's concept-formation program (Winston, 1975) and Mitchell's developments of this (Mitchell, 1978), plus Michalski's work on the INDUCE systems (Michalski, 1983), where the inductive inference method involved generalising variables and dropping conditions.

LEX system (Mitchell *et al*, 1983), plus work done by G. Plotkin, M. Thomas and others, as discussed below. Since 1980, Mitchell, Utgoff, Nudel and Banerji have developed a system called LEX which acquires heuristics in order to solve symbolic integration problems. It does this by the procedure of generating a practice problem, using available heuristics to solve this problem, and then analysing the search steps used to obtain the solution. New (domain-specific) heuristics are proposed to improve the performance on subsequent problems, and are tested out on suitable generated examples.

More specifically, LEX starts off knowing a set of operators to try, and when they may be applied, with the goal of deriving heuristics about when application of the operators is appropriate. LEX uses the idea of version spaces by keeping track of all possible versions of a heuristic from the most specific to the most general; these versions are related by ISA-hierarchy generalisation. As new problems are tried, and more positive and negative instances of rule application are encountered, the gap narrows between the most general and most specific versions permitted by those instances; in fact it is sufficient to keep track only of these boundaries (ie. the most specific and the most general versions that have not been ruled out). In addition, LEX can propose problems that will help to narrow the version space, by providing crucial experiments that distinguish between more general and more specific cases. See (Boswell, 1987) for an assessment of the later development of the system, LEX2.

Further references for inductive generalisation include Mitchell's article "Generalization as Search", which compares existing programs that generalise from examples, with respect to the standpoint of the title (Mitchell, 1982a) and Plotkin's paper "A Note on Inductive Generalization" (Plotkin, 1969). Other relevant work in this field includes that by Kinber and Brazma to formalise the process of generalisation (Kinber & Brazma, 1990), Muggleton's work on logic programs (Muggleton, 1988) and Thomas and Jantke's generalisation of rewrite rules during the course of inductive completion (Thomas & Jantke, 1989). Kinber and Brazma formalise the process of generalisation of sample computations explaining the behaviour of a program, in order to synthesise such a program: detection of fragments

of arithmetical progressions may be made via a system of inductive inference rules (Kinber & Brazma, 1990). Muggleton has incorporated the production of the least general generalisation of given predicates ("predicate invention") into the Inductive Logic Programming tool Golem (Muggleton, 1988). For example, this allows the construction of *father-or-father-in-law* from examples of *grandfather* and background about *mother*. Although others (eg. Banerji) have described mechanisms for predicate invention, Muggleton's research is the first to provide a theoretical framework for predicate invention; this is carried out within the lattice of entailment relative to interpretation rewrites. However, the implementation cannot yet deal with inventing a partition for the "quick-sort" algorithm.

Muffy Thomas has tackled methods of generalising infinite sets. The Knuth-Bendix procedure (see Subsection 3.2.4) generates a decision procedure for equality, but may diverge and generate an infinite set. Her approach is based on finding exact generalisation of the varying parts of the infinite sequence of rules, so that the infinite sequence of rules may be replaced by an equivalent finite sequence. Sometimes the solution is to introduce new subtypes, and deduce a new (enriched) order-sorted signature. All the terms of the introduced sort are those over which generalisation should be made; the addition of the corresponding rule results in a confluent and terminating system (Thomas & Watson, 1991). This gives an effective procedure for generalisation, which sometimes works, but is restricted to a regular tree language.

Relevant work in the field of inductive learning has been discussed. However, all of these approaches are limited, and there is little work being carried out which is analogous to the process described in this thesis of one logical system guiding the construction of a proof in another.

3.4 Explanation-Based Generalisation

This section gives a brief description of explanation-based generalisation, since the notion is used elsewhere in this thesis. In particular, it is involved in the method proposed for generalisation of proofs in Chapter 8. Explanation-based generalisation is a technique for formulating general concepts on the basis of specific training examples, first described in (Mitchell *et al*, 1986). For example, a particular solution may be generalised to obtain a specification of the most general problem which it solves. Typically, such methods require one algorithm to perform a generalisation from an old domain, another to determine the applicability or otherwise of the generalisation to a new domain, and yet another to actually apply the generalisation in the new domain. In fact, unification alone may be used for the initial stage of constructing a generalised solution. The information about unification is fed through various stages of the construction using back propagation, in a manner described for example in (Boswell, 1987, P77) (with reference to LEX2). The construction of patterns for problem-solving within LEX2 is illustrated in Figure 3–1, in which constraint back-propagation is used to enable explanation-based generalisation. With reference to this figure, the most general form of the expression for which the rules would be applicable may be calculated as follows:

- The form of an expression to which *Rule1* is applicable must be $\int S.G(x)dx$, and afterwards $S.\int G(x)dx$, for any rational number S and first-order function G.

- For *Rule2* to be applicable, $G(x)$ must be $x^{K\neq -1}$, for any rational number K.

- Any such information derived will then be filtered back to the original expression, to give a most general generalisation (using the rules used on the original problem) of $\int S.x^{K\neq -1}dx$.

Original Problem	Rule Used	Generalisation
$\int 5.x^2 dx$		$\int S.x^{K \neq -1} dx$
$\Downarrow$	$Rule1 : \int R.F(x)dx \Rightarrow R. \int F(x)dx$	$\Uparrow$
$5. \int x^2 dx$		$S. \int x^{K \neq -1} dx$
$\Downarrow$	$Rule2 : \int x^{R \neq -1} dx \Rightarrow \frac{x^{R+1}}{R+1}$	$\Uparrow$
$5.\frac{x^3}{3}$		$S. \int G(x)dx$

Figure 3–1: An Example of Explanation-Based Generalisation in LEX2

Kedar-Cabelli and McCarty have developed a Prolog meta-interpreter called PROLOG-EBG which constructs generalisations from training examples (Kedar-Cabelli & McCarty, 1987). Donat and Wallen have developed this work within Isabelle to provide a generalisation method based on higher order unification and resolution (Donat & Wallen, 1988). In addition, it has been argued that explanation-based generalisation is equivalent to the technique of partial evaluation, used in functional and logic programming as a program optimisation method (van Harmelen & Bundy, 1987).

See Subsection 8.1.4 for specific discussion of explanation-based generalisation methods with regard to the new domain of general proofs described in Chapter 6.

3.5 Primitive Recursion

This section discusses the possibility of an extension of the notion of primitive recursion which will be further considered in Chapter 5.

The class of primitive recursive functions is the smallest class (of functions $\mathbb{N}^k \rightarrow \mathbb{N}$) containing the basic functions of zero, successor and projection, and closed under the basic operations given below (where d is a constant and h is some primitive recursive function:

Recursion: (with optional arguments $\vec{x}$) for g, h already defined as primitive re-

cursive, then so is f where:

$$f(0,\vec{x}) = g(\vec{x}); \quad f(n+1,\vec{x}) = h(f(n,\vec{x}),n,\vec{x})$$

Composition: If $g : \mathbb{N}^j \to \mathbb{N}$ and $h_i : \mathbb{N}^k \to \mathbb{N}$ $(1 \leq i \leq j)$ are primitive recursive, then by composition so is $f : \mathbb{N}^k \to \mathbb{N}$ defined by

$$f(n_1,\ldots,n_k) = g(h_1(n_1,\ldots,n_k),\ldots,h_j(n_1,\ldots,n_k)).$$

The recursive functions are those obtained from these functions and rules, plus those obtained by use of the least number operator ($f(\vec{x}) \equiv$ the least number $n \in \mathbb{N}$ such that the primitive recursive function $g(\vec{x},n)$ is 0). Some functions, such as Ackermann's function, are computable (and hence recursive by Church's thesis), but not primitive recursive.

Tucker, Zucker and Wainer have generalised the notion of primitive recursiveness to that of "CR-computability" (and also the notion of recursive enumerability[8] to "semicomputability") (Tucker *et al*, 1990, P10).[9] They define, for each $A \in K$, where K is a class which is closed under isomorphism (also called an abstract data type), and A is a Σ-structure, and various other restrictions, functions f_i of type $(\mathbb{N},\vec{k};l_i)$ such that for $i = 1,\ldots,m$,

$$f_i(0,\vec{x}) = g_i(\vec{x})$$

and for $z > 0$,

$$f_i(z,\vec{x}) = h_i(z,\vec{x},f_1(\hat{\delta}_1(z,\vec{x}),\vec{x}),\ldots,f_m(\hat{\delta}_1(z,\vec{x}),\vec{x}),\ldots$$

$$\ldots,f_1(\hat{\delta}_d(z,\vec{x}),\vec{x}),\ldots,f_m(\hat{\delta}_d(z,\vec{x}),\vec{x}))$$

where for $i = 1,\ldots,m$, g_i and h_i are CR-computable (of suitable type), and for $i = 1,\ldots,d$, $\hat{\delta}_i$ is defined by

$$\hat{\delta}_i(z,\vec{x}) = min(\delta_i(z,\vec{x}),z-1),$$

---

[8]A set is recursively enumerable if and only if there exists some primitive recursive function Ψ such that $A = \{y|\exists x\Psi(x) = y\}$.

[9]Note also the use of starred sorts (as given in (Tucker *et al*, 1990, P5)) to combat the lack of effective coding of finite sequences within abstract structures in general.

for some CR δ_i of type $(\mathbb{N}, \vec{k}; \mathbb{N})$, so that for $z > 0$, $\hat{\delta}_i(z, \vec{x}) < z$. Over $\mathbb{N}$, all these schemes are equivalent to simple primitive recursion, as with the cases which I shall consider in Chapter 5. Note the type of f_i: this definition is not suitable for the basic types which I wish to consider which are strings, labelled trees, lists and (Cartesian) products of these types, and for which the recursion is over a different data structure than the natural numbers. Hence I shall define in Section 5.1 an analogous notion of primitive recursive form for datatypes other than the natural numbers, and call this "effectiveness".[10]

3.6 Summary

The chapter is devoted to the provision of background and relevant definitions for notions which shall be referred to in later chapters. Much interest and contemporary work is being carried out in the fields discussed in this chapter, and one may conclude that there is plenty of scope for exploiting their relationships and using some as a means for effecting advances in others.

[10]This should not be confused with the related well-known notion of effective computability (see (Boolos & Jeffrey, 1980)).

Chapter 4

Formalisations of Arithmetic

$1 \in N.$
$a \in N. \supset .a+1 \in N.$
$a, b \in N. \supset : a = b. = .a+1 = b+1.$
$a \in N. \supset .a+1- = 1.$
$k \in K .\therefore 1 \in k .\therefore x \in N. x \in k :\supset_x x+1 \in k ::\supset .N \supset k.$

Giuseppe Peano

This chapter presents a formal description of a representation of first order Peano Arithmetic in Section 4.1, which is reliant upon the description of sequent calculus given in Appendix A, followed by a description of the semi-formal system of first order arithmetic with the ω-rule. It proceeds to discuss first the need and then the possible methods of restriction of such a rule. In Subsection 4.3.3 the chosen approach for restriction of the ω-rule is detailed.

4.1 PA: First Order Peano Arithmetic

At the head of the chapter Peano's five postulates were given, where $\supset$ denotes implication. These postulates are a set of axioms for the natural numbers which were made explicit before formal systems for the latter were studied, and are still central to the modern formulation of arithmetic: *classical arithmetic* is taken to be the theory of the natural numbers built up from Peano's axioms, classical pred-

icate logic and the use of primitive recursive definitions. As discussed in Subsection 2.1.1, *intuitionist arithmetic* differs from classical arithmetic in its underlying philosophy: in particular, statements are understood in terms of their assertability, rather than their truth. The axioms of Peano Arithmetic are expressed in a first-order language of arithmetic, and may directly be seen to be true from our grasp of the fundamental nature and structure of the natural numbers. The latter is expressed in our understanding of the natural numbers as the smallest structure closed under a one-one mapping and containing an element which does not lie in the range of that mapping.

Syntactical details about the formal system PA follow.[1] A formal system is comprised of a language, axioms and rules of inference. The language normally consists of infinite but denumerably many symbols, finite sequences of which form expressions. Certain expressions (namely the meaningful ones) are designated as the formulae of the language. The formulae of PA (and also PA_ω) are those of formal number theory.[2] The latter will be briefly outlined at this stage to determine the conventions to be used. The symbols of the language will be variables, n-ary function symbols (a 0-ary function symbol is a constant) and n-ary predicate symbols for each n (the binary predicate symbols including equality), plus the logical symbols $\neg, \vee, \wedge, \rightarrow, \forall$ and $\exists$. (In fact, one only needs to have $\neg, \vee$ and $\exists$, and could define the other logical symbols in terms of these). Parentheses and commas indicate grouping. Terms are formed from 0 and number variables by means of the successor function s, $+$, . and functions. Atomic formulae are of the form $x = y$, where x and y are terms. Formulae are built up from terms in the normal way, using quantification on number variables only, to give a first order theory. So, formulae are constructed by a generalised inductive definition such that

1. an atomic formula is a formula;

2. if u is a formula then $\neg u$ is a formula;

---

[1] See for example (Buchholz & Wainer, 1987) for an alternative formulation.

[2] As given in for example (Schwichtenberg, 1977).

3. if u, v are formulae, then $u \vee v$, $u \wedge v$ and $u \rightarrow v$ are formulae;

4. if u is a formula and x is a variable then $\exists x\ u$ and $\forall x\ u$ are formulae.

The axiomatisation of both systems PA and $PA_{c\omega}$ will be given in terms of sequents,[3] that is expressions of the form $\Gamma \vdash \Delta$, where Γ and Δ are finite (possibly empty) lists of formulae. In fact, I shall use a representation where there is just a single conclusion in the sequent (that is, Δ must be a formula). Defining PA in terms of such intuitionistic variants enables the definition of HA, the intuitionistic counterpart of PA, merely by removing the law of excluded middle. The derivations in PA may be regarded as being in tree form, with a given analysis as in Appendix A, and the proofs as being derivations with axioms at the leaves. The rules of inference for PA are the standard rules for the logical connectives[4] and equality, the axioms are the schema $\Gamma \vdash A$, where $A \in \Gamma$, plus the axioms of arithmetic, and also there is the rule of induction. For reference, Peano's axioms and the chosen form of the rule of induction under consideration (for there are many forms) are stated below.

4.1.1 Peano's Axioms

$\vdash \forall x\ \forall y\ s(x) = s(y) \rightarrow x = y;$	*1-1 property of successor function*
$\vdash \forall x\ \neg(s(x) = 0);$	*0 is not the successor of anything*
$\vdash \forall x\ x + 0 = x;$	*addition is primitive*
$\vdash \forall x\ \forall y\ x + s(y) = s(x + y);$	
$\vdash \forall x\ x.0 = 0;$	*multiplication is primitive*
$\vdash \forall x\ \forall y\ x.s(y) = (x.y) + x.$	

These axioms include a convenient representation of addition and multiplication (although these functions can be defined in terms of the successor function using

---

[3] See (Prawitz, 1971, P262–266) for a natural deduction-style formulation of first-order Peano arithmetic, and see Appendix A for a description of sequent calculus.

[4] See Table A–1.

primitive recursion). In a similar way, defining equations for each elementary function may be given. Induction, substitution, the concept of the natural numbers as being generated from an initial member via the sucessor function, together with the first two axioms given above correspond to Peano's five Postulates given at the start of this chapter.

Note that a free variable formulation of these rules is used in Seldon[5]. Regarding the question as to whether the axioms of PA should be formulated in a quantifier-free form, rather than as above: if a free variable formulation had been given, instantiation would be a derived rule, but this would involve use of the substitution rule (see Subsection 4.1.5), which requires cut if it is not to be taken as a basic rule. However, the use of quantification theory in proofs of quantifier free formulae can be eliminated (Kreisel, 1965, P161).

4.1.2 Induction Rule

This is the version used in the implementation:

$$\frac{\Gamma \vdash A(0) \qquad \Gamma \vdash \forall y(A(y) \rightarrow A(s(y)))}{\Gamma \vdash \forall x A(x)}$$

An alternative representation often used which may be derived from this is:

$$\frac{\Gamma \vdash A(0) \qquad \Gamma, A(r) \vdash A(s(r))}{\Gamma \vdash A(y)}$$

Peano's induction rule is obtained as an instance of the generalised (Noetherian) induction rule:

$$\frac{\forall m \in M(\forall k \in M\ k <_M m \rightarrow \Phi(k)) \rightarrow \Phi(m)}{\forall m \in M\ \Phi(m)}$$

where $<_M$ is a well-founded order of the set M.

4.1.3 Basic Sequents Involving Identity

1. $\vdash s = s$ for all terms s.

---

[5]The implementational system — see Chapter 10.

2. $s_1 = t_1, \ldots, s_n = t_n \vdash f^n(s_1, \ldots, s_n) = f^n(t_1, \ldots, t_n)$ for f any function letter, s_i, t_i any terms.
   Hence $t_1 = t_2 \vdash s(t_1) = s(t_2)$.

3. $s_1 = t_1, \ldots, s_n = t_n, R^n(s_1, \ldots, s_n) \vdash R^n(t_1, \ldots, t_n)$ for every predicate constant, including '='.
   Hence $s = t, s = r \vdash t = r$.

4. $s = t \vdash t = s$ for all terms s and t.

5. $s = t, t = r \vdash s = r$ for all terms s, r and t.

1.–3. are the equality rules normally used (for example see (Takeuti, 1987, P69)), but as Gallier has shown, these only allow elimination of "essential" (ie. non-atomic) cuts (Gallier, 1986); 4. and 5. are needed to allow the elimination of atomic cuts (cf. natural deduction representation given in (Kaye, 1991, P117)). Thus, so long as the equality sequents above are taken as axiomatic, it is the presence of the induction rule (for which cut reduction only carries through as far as Σ_1-formulae), rather than equality statements, which prevents cut elimination holding for PA.

4.1.4 The Cut Rule

Subsection 2.3.1 has already introduced various forms of the cut rule. In Seldon[6] the cut rule is given in a form appropriate for backward reasoning as:

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C} \tag{4.1}$$

The repercussions of this modification will be discussed at a later stage in Chapter 10. If the rule of repetition is used in order to allow repeated formulae, rule 4.1

[6]The system used for implementation: see Section 10.2.

may be derived using 2.2:

$$\dfrac{\dfrac{\Gamma, \vdash A \qquad \Gamma, A \vdash C}{\Gamma, \Gamma \vdash C}\,2.2}{\Gamma \vdash C}\,\text{rep repeated (for all formulae of } \Gamma)$$

Alternatively,

$$\dfrac{\dfrac{\Gamma \vdash A}{\Gamma, \Delta \vdash A}\,rthin \qquad \dfrac{\Delta, A \vdash C}{\Gamma, \Delta, A \vdash C}\,rthin}{\Gamma, \Delta \vdash C}\,4.1$$

This shows that, in a system which includes the rules of repetition and thinning, rules 2.2 and 4.1 are equivalent.

4.1.5 The Rule of Substitution

Note that the rule of substitution[7]

$$\dfrac{\Gamma \vdash A = B \quad \Delta \vdash R[A]}{\Gamma,\ \Delta \vdash R[B]}\,subst$$

is a derived rule of inference, using axiom 3. from Subsection 4.1.3, and the cut rule (twice), as shown below:

$$\dfrac{\Delta \vdash R[A] \qquad \dfrac{\Gamma \vdash A = B \quad \dfrac{}{A = B, R[A] \vdash R[B]}\,axiom}{\Gamma, R[A] \vdash R[B]}\,cut(A = B)}{\Gamma,\ \Delta \vdash R[B]}\,cut(R[A])$$

However, as mentioned above, using substitution as a derived rule would mean that the cut rule is needed for very many proofs. Taking substitution as a basic rule, as in the implementation, avoids this problem.

---

[7]Note the conventional use of $R[A]$ to signify that A has been substituted for some occurrence of x in $R(x)$.

4.2 PA_ω: First Order Arithmetic with Infinite Induction

In this section a system of arithmetic with the ω-rule shall be described, with a view to implementation. This will have infinite proofs and therefore not be a formal system in the usual sense. As discussed in Chapter 2, such a system may have various desirable properties such as cut elimination; see (Prawitz, 1971, P266–267) for a natural deduction representation, for which (analogously) full normalisation may be carried out.[8]

The system PA_ω is essentially PA enriched with the ω-rule in place of the rule of induction.[9] The derivations are then infinite trees of formulae; a formula is demonstrated in PA_ω by "exhibiting" a prooftree labelled at the root with the given formula. I shall consider later how the prooftrees for the system $PA_{c\omega}$ might be described, but first syntactical details about this system PA_ω will be given (cf. (López-Escobar, 1976, P162)). The formulae of PA_ω are those of formal number theory, with the restriction that the sequents of PA_ω consist exclusively of sentences (that is, closed formulae).[10] The logical rules of inference for the

---

[8]Cut-free derivations in the sequent calculus correspond to normal derivations in natural deduction systems. If the natural deduction derivation of a sequent is normal, one can find directly a cut-free proof of the corresponding sequent (Prawitz, 1965, P88-93). See also the discussion in Subsection 3.1.1.

[9]See Subsection 4.1.2.

[10]Attention may be restricted to closed formulae only (ie. ones without any free variables), since the free variable x in the premise of the $\forall r$-rule is now replaced by infinitely many premises $\Gamma \vdash A(n)$ in the ω-rule. The restriction will ensure that every free term is numerical; moreover, the axioms and basic inferences will be restricted to formulae in which no free number variables occur.

A problem arises due to the fact that there are free variables in the implementational system, and free variables in PA, but no free variables in $PA_{c\omega}$ as it is presented in this chapter. The implementation approach is to retain the $\forall l$-rule, and add the ω-rule (although it could be stipulated that the $\forall r$- and $\exists l$-rules should not be used). The theoretical results state that PA_ω or $PA_{c\omega}$ are complete for sentences, and make use of the fact that the ω-rule only applies to sentences (cf. Section 2.5). The closed

propositional calculus are the same for PA_ω as for PA, as are the structural rules of inference. The rules of inference for the quantifiers differ: $\forall l$ and $\exists r$ are the same, but the following replace $\forall r$ and $\exists l$:

$$\frac{\Gamma \vdash A(0) \quad \Gamma \vdash A(\underline{1}) \ \ldots \ \Gamma \vdash A(\underline{k}) \ldots}{\Gamma \vdash \forall x A(x)} \forall r_\omega$$

$$\frac{A(0), \Gamma \vdash \Delta \quad A(\underline{1}), \Gamma \vdash \Delta \ \ldots \ A(\underline{k}), \Gamma \vdash \Delta \ \ldots}{\exists x A(x), \Gamma \vdash \Delta} \exists l_\omega$$

There is no logical axiom schema, but there is an arithmetical axiom schemata: For closed terms t_1, t_2,

If $t_1 = t_2$ under canonical interpretation, $\vdash t_1 = t_2$ is an axiom of PA_ω.

If $t_1 \neq t_2$ under canonical interpretation, $t_1 = t_2 \vdash$ is an axiom of PA_ω.

In certain formulations $t = u, A(t) \vdash A(u)$ would be added to the axioms for equality, but this is already covered by the representation of PA given in Section 4.1 (Girard, 1987, P349).

PA_ω has been described by Schütte as a semi-formal system to stress the difference between this and usual formal systems which use finitary rules, for PA_ω has basic inferences with infinitely many premises (namely the ω-rule) (Schütte, 1977, P174).

---

formula restriction is basically made to avoid problems arising when there is nesting of free variables; thus, a restriction to the examples considered in Chapter 8 such that free variables are not used in these examples should avoid any problems due to the fact that free variables may be carried across in the implementation. The consequences of possibilities of use of free variables in the implementation have not been checked, but presumably so long as this restriction is carried out, there should not be a problem.

4.3 $PA_{c\omega}$: First Order Arithmetic with the Constructive Omega Rule

4.3.1 The Use of a "Constructive" Rule

As discussed in Section 2.1, for implementational purposes infinite proofs must be thought of in the constructive sense of being generated, rather than absolute. So, having described infinite prooftrees in the metatheory given above, I would like to restrict attention to the constructive prooftrees. In Chapter 5 a characterisation of effective infinite proofs is given, to allow presentation of the system PA enriched with a constructively restricted ω-rule in place of the rule of induction. In order to do this, the question of whether the 'recursive' or rather the 'primitive recursive' ω-rule should be used is now considered (in the sense of appropriate restrictions on the ω-rule which are to be specified).

It is important to consider whether it would be preferable to generate the derivations by means of a recursive, or else a primitive recursive, function, and hence whether this corresponds to $PA_{r\omega}$ or the analogous system $PA_{c\omega}$ ($PA+$ primitive recursive restricted ω-rule \ induction). This will depend upon whether it is the recursive ω-rule or its primitive recursive counterpart which will provide the basis for implementation, which will be decided by whether the general proof representation (in terms of rewrite rules) is recursive or primitive recursive.

However, if one has the rule of repetition $\frac{\Gamma\vdash\Delta}{\Gamma\vdash\Delta}$ in PA_ω, any recursive derivation can be "stretched out" to a primitive recursive derivation using the same rules of inference, plus this rule. Indeed, the cut elimination theorem does not hold for $PA_{c\omega}$ if this rule is omitted (López-Escobar, 1976, P169). Since if the derivation Π is recursive, and the rule of repetition is allowed, then there is a primitive recursive derivation Π' not equal to Π such that Π and Π' prove the same formula, one might as well allow the rule of repetition, and just concern oneself with primitive recursive definitions. Although the rule of repetition may be used if necessary, its use will not normally be a useful option under a goal-directed discipline such as

the implemented system, and it must be used at most a finite number of times in any branch, in order to prevent looping. In the next chapter I shall show that the general proof notation in terms of application of rewrite rules may be represented effectively (where effectiveness will be presented as an analogous concept to primitive recursion).

What are the properties of the resulting system? As mentioned in Chapter 2, Nelson has shown completeness for PA with the primitive recursive ω-rule in place of induction (Nelson, 1971). $PA_{r\omega}$ is no stronger than $PA_{c\omega}$ (see Chapter 4), since it has the same syntax.

However, intuitionistic arithmetic is a stronger theory with the recursively restricted ω-rule than with the primitive recursively restricted ω-rule (but it is also a stronger theory with the ω-rule than the recursively restricted ω-rule) (Löpez-Escobar, 1977, P77). As mentioned above, this is not so if the law of excluded middle is added, thus giving a classical system; see Section 2.5 for further details.

Note that there is also an analogy to be drawn between my implementational approach and finitism. Formal systems of recursive number theory, in which generality is expressed by free variables and existence by the actual presentation of an instance or (if the object depends on parameters) a function, may be said to be finitary if the functions admitted are sufficiently elementary — for example, primitive recursive functions. In such formalisms any formula will express a general statement, each instance of which can be checked by computation. For this reason classical logic can be used. Hence there is an analogy with finitist methods regarding my use of the ω-rule, and also this justifies use of the (classical) system $PA_{c\omega}$, which has been generated in a constructive manner.

In conclusion, I shall choose the (classical) system PA with a primitive recursive restriction on the prooftrees. This is because my approach is not that of using reflective methods, and moreover because there is cut elimination in this system, it is suitable for implementation and also PA with the unrestricted ω-rule forms a conservative extension of this system.

4.3.2 Methods of Restriction

This subsection presents the usual approach to making the ω-rule effective. The normal approach when dealing with a system with infinitary proofs such as PA_ω is to work with codes for the derivations[11] rather than using the derivations themselves, where the codes are natural numbers, and are defined inductively, corresponding to the inductive build up of such derivations.[12] The inductive definition carries through easily for the finite rules, but for the ω-rule there is a problem as in general there are infinitely many premises. The solution normally adopted is to assume that the codes for the premises can be enumerated by a primitive recursive function, and to use a code (or primitive recursive index) of such an enumeration function to construct a code of the whole derivation (see for example (Girard, 1987, P390)). The code of a derivation should contain sufficient information about the derivation, such as allowing the primitive recursive inference of the name of the conclusion of the last inference of the derivation, a bound for the length of the derivation, a bound for the cut-rank, etc. See (Schwichtenberg, 1977, P886) for further details, including the case of the ω-rule. All such derivations obtained by embedding arithmetic into this infinite system can be coded. So, as discussed in Section 2.5, the restriction on the primitive recursive ω-rule is that there is a primitive recursive f for which, for every n, $f(n)$ is the Gödel number of the proof of $A(\underline{n})$, the nth numerator of the ω-rule (and analogously for the recursive ω-rule).

If infinite PA_ω-derivations are considered as well-founded trees in this manner, then any code u of a PA_ω-derivation d can be thought of as being obtained in an inductive manner by a code of the corresponding subderivation being attached to each node of the tree corresponding to d. Hence the property of u being a code for a PA_ω-derivation (cf. definition given in (Schwichtenberg, 1977, P886)) is

---

[11]Note that, if numeric encoding is used, statements like $\ulcorner \forall x\ x = 0 \urcorner = 1377890425$ are provable, whereas such expressions will not be present in the system which I utilise.

[12]See Section 2.4.

equivalent to u being associated with such a well-founded tree. This notion may be generalised (Schwichtenberg, 1977, P894):

> "One has to express that at any node (= sequence number) n the tree is locally correct, ie. that the code u_n affixed there (u_n can be easily defined by induction on n) and all its predecessors $u_{n*(i)}$, $i = 0, 1, 2, \ldots$ fulfil a relation as given in the definition of codes for ($PA_{c\omega}$-derivations). The well-foundedness is then obtained automatically, since in particular $|u_{n*(i)}| \leq |u|$ is required and $\leq$ is a well-ordering."

I shall not use this approach to making the ω-rule effective, but shall instead adopt a different method, which is described in the following subsection.

4.3.3 The Chosen Approach

In this subsection further discussion is provided about placing a restriction on the prooftrees of PA_ω such that only those which have been constructively generated are allowed, in order to capture the notion of infinite labelled trees in a finite way. The chosen approach is that constructive proofs to which attention is restricted shall be characterised directly. The meaning of the restriction that the numerators of the ω-rule may be effectively generated will be discussed in this subsection. In Chapter 5 I shall present a representation of $PA_{c\omega}$ in terms of effectively-given ω-prooftrees, which may not be defined until the notion of primitive recursion over datatypes other than the natural numbers has been expressed in Section 5.1. Before this, the nature of restrictions on the ω-rule which render it suitable for implementation must be considered.

Kreisel discusses the question of appropriate restriction of the proofs of PA_ω as follows (Kreisel, 1965, P163):

> "What restrictions should be put on the proof figures ...? First, the obvious way of ensuring inclusion of the new system in the old is that one should be able to "talk" about the new system in the old. Precisely, each proof figure is a partial ordering of formulae with premises preceding conclusions: it should be definable in the old system, and it should be provable that (the formula at) any node is related to its immediate predecessors according to the rules."

Thus, by adding the provability relation and numeric encoding, a reflection system which necessarily extends the original one may be formed (see Chapter 2 for a comparison between reflective systems and semi-formal systems with the ω-rule). However, the necessity of using this Gödel numbering approach may be avoided by following Tucker in defining primitive recursion over various data-types (Tucker *et al*, 1990), and also Feferman (Feferman, 1989). This is the approach that I shall take.

If an arithmetical encoding method were to be used, the primitive recursive constraint could be attached directly to the ω-rule. Recall that the coding restriction would be that there is a (primitive) recursive function f such that for each natural number n, $f(n)$ is the code of a (possibly infinite) derivation of $P(\underline{n})$. However, without using such an approach it is not possible to place the restriction on when the rule is applied, and instead it must be placed on the shape of the proof in which the ω-rule appears. So in my case, the primitive recursive restriction on the ω-rule is that what is restricted is the shape of the trees: only derivations which are "effective" will be accepted, a notion which must be defined.

In the previous subsection a notion of PA_ω was given, but there remained the question of giving some restriction on the proofs involving the ω-rule. I now define
$\vdash_{PA_{c\omega}} \Phi$ *iff* $\exists f.$ f is an 'effective' prooftree of $PA_{c\omega}$ with Φ as initial sequent
This is a definition of $PA_{c\omega}$, but it does not define $PA_{c\omega}$ as a (semi)-formal system in the sense that it does not say what the axioms and rules of inference are. This definition replaces the usual approach of using numeric encoding (using notation $\ulcorner \urcorner$) to consider some statement used to strengthen PA of the form:
$\vdash_{PA} (\exists \Pi\ (\mathsf{prooftree}(\Pi) \wedge \mathsf{conc}(\Pi) = \ulcorner \Phi \urcorner)) \rightarrow \Phi$

In conclusion, it is possible to place restrictions on the admissible prooftrees without using Gödel encoding, but this necessitates the clarification of what it is to place restrictions on the subproofs themselves.

4.4 Conclusions

The chosen approach for restriction of the ω-rule such that it may be implemented has been presented. In order to carry out this approach without using numeric encoding, a new notion of effectiveness must be defined. This will form the subject of the following chapter.

Chapter 5

Effective Prooftrees for $PA_{c\omega}$

"When one's proofs are aptly chosen, four are as valid as four dozen."

Matthew Prior

In this chapter a characterisation of $PA_{c\omega}$, which is a representation of arithmetic with the ω-rule restricted in the way described in the previous chapter, is presented in terms of "effective" prooftrees. In order to do this, the notion of effectiveness (using primitive recursive definitions in theories other than the natural numbers) is defined.

5.1 Effectiveness over Various Datatypes

In this section I shall consider the question, and give appropriate definitions, of what it is to be a primitive recursive function for a function which does not necessarily have domain $\mathbb{N}$. The analogous property shall be denoted "effectiveness", and bears a strong relation to structural recursion in versions of constructive type theory and functional programming languages. This section closely follows Tucker et al., whose work on this topic is discussed in Section 3.5. The type of f_i given in Tucker's definition means that this definition is not suitable for the basic types which I wish to consider, which are strings, labelled trees, lists and (Cartesian) products of these types, and for which the recursion is over a different data structure than the natural numbers. Hence I shall define an alternative generalisation

of primitive recursion for datatypes other than the natural numbers, and call this "effectiveness".

In order to define effectiveness for a particular datatype, it is necessary to define the basic functions, composition and recursion, and then state that the effective functions over that datatype is the smallest set closed with regard to these operations. Certain patterns will be general, and will be considered before the more specific cases.

I shall define what it is to be effective for functions $f : S_1 \times S_2 \ldots \times S_k \rightarrow S_{k+1}$, where S_i (for $1 \leq i \leq k+1$) is one of the following sorts: *string* (representing sequents) or *nat*; or else formed using type constructors: S_i *list* (representing lists of arbitrary sort; in particular, *nat list* represents tree-positions). In the next section I shall consider representation of trees labelled with objects of type S_i. This is with the ultimate goal of considering in a further section whether or not the prooftrees for $PA_{c\omega}$ are effective.

In the following sections, the notation will be employed that *type* (or $type_i$ for $i \in \mathbb{N}$) may range over sorts.

General

Projection functions: $q_i^k(A_1, \ldots, A_k) = A_i \quad q_i^k : type_1 \times \ldots \times type_k \rightarrow type_i$

Composition: If $g : type_1 \times \ldots \times type_j \rightarrow type, h_i : type_{S_1} \times \ldots \times type_{S_k} \rightarrow type_i, 1 \leq i \leq j$ are effective, then $f : type_{S_1} \times \ldots \times type_{S_k} \rightarrow type$ is primitive recursive, where for $Str_i \in type_{S_i}$:

$$f(Str_1, \ldots, Str_k) =_S g(h_1(Str_1, \ldots, Str_k), \ldots, h_j(Str_1, \ldots Str_k))$$

Specific

In all of the following cases, the effective functions are the basic functions and projection functions, together with all those which may be generated from effective functions by a finite number of applications of the rules of composition and recursion. The definition of datatypes shall be given in terms of constants and constructor functions.

5.1.1 Natural Numbers: $f : nat \rightarrow type$

If *type* is *nat*, this is just primitive recursion.

Datatype: $nat = \{0\} \cup \{s(n) | n \in nat\}$

1. the constant is 0

2. the constructor function is the successor function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $s(x) = x + 1 \quad \forall x \in \mathbb{N}$, as with primitive recursion.

Recursion:

If $g : type_1 \times \ldots \times type_k \rightarrow type$ and $h : type_1 \times \ldots \times type_k \times nat \times type \rightarrow type$ are effective, then so is $f : type_1 \times \ldots \times type_k \times nat \rightarrow type$, where:

$$f(n_1, \ldots, n_k, 0) = g(n_1, \ldots, n_k)$$

$$f(n_1, \ldots, n_k, s(x)) = h(n_1, \ldots, n_k, x, f(n_1, \ldots, n_k, x))$$

(the $n_i \in type_i$ may be omitted).

5.1.2 Lists: $f : t\ list \rightarrow type$

Note that the elements of the list may be of any type, and not just *nat*.

Datatype: $t\ list = \{[\,]\} \cup \{cons(x, y) | x \in t, y \in t\ list\}$

1. the constant is [ ] ($\equiv nil$)

2. constructor function is "cons"

   $cons : type \times type\ list \rightarrow type\ list$, written ::.

Recursion:

If $g : type_1 \times \ldots \times type_k \rightarrow type_m$ and $h : type_1 \times \ldots \times type_k \times type_n \times type_n list \times type_m \rightarrow type_m$ are effective, then so is $f : type_1 \times \ldots \times type_k \times type_n\ list \rightarrow type_m$, defined by

$$f(n_1, \ldots, n_k, [\,]) = g(n_1, \ldots, n_k)$$

$$f(n_1, \ldots, n_k, H :: T) = h(n_1, \ldots, n_k, H, T, f(n_1, \ldots, n_k, T))$$

(where the $n_i \in type_i$).

(Note that functions such as $head : type\ list \rightarrow type$ and $tail : type\ list \rightarrow type\ list$ are definable using recursion.)

5.1.3 Strings: $f : string \rightarrow type$

In this subsection a definition of effectiveness for strings is given, where strings are taken to be binary trees with characters at the nodes, used to represent the syntax of formulae, etc. This representation allows a concatenated expression to be split uniquely into two parts, which is necessary when making the recursive definition below. Suppose that a finite set of characters (C) is given.

Datatype: $string = C \cup \{st\hat{*}st | st \in string\}$

1. constant strings are taken to be the initial characterisation of characters.
2. the constructor function is $\lambda x.\ \lambda y.\ x\hat{*}y$ (syntactical juxtaposition or concatenation — see for example (Kleene, 1962, P71)), where $\hat{*} : string \times string \rightarrow string$ and 'A'$\hat{*}$'B' $=_S$ 'AB'.

Recursion:

Given $g_{char} : type_1 \times \ldots \times type_k \rightarrow string$, $h : type_1 \times \ldots \times type_k \times string \times string \times type \rightarrow type$ which are effective, then $f : type_1 \times \ldots \times type_k \times string \rightarrow type$

is primitive recursive, as defined below, where $S_1, \ldots, S_k$ ($S_i \in type_i$) may be omitted:

$$f(S_1, \ldots, S_k, \text{'}char\text{'}) =_S g_{char}(S_1, \ldots, S_k)$$

$$f(S_1, \ldots, S_k, A\hat{*}B) =_S h(S_1, \ldots, S_k, A, B, f(S_i, \ldots, S_k, A), f(S_1, \ldots, S_k, B))$$

(The characters are a finite, enumerated set, so definition by cases is effective.)

5.2 Labelled Trees

In the previous section I have defined what it is for $f : t_1 \times \ldots \times t_n \rightarrow t_m$ to be effective, where $t_i = string$ or nat or $type\ list$. To define labelled trees, one may just consider $t : nat\ list \rightarrow label\_type$: it is not necessary to define an extra datatype *tree*. The intention in this section is to represent ω-prooftrees as $f : nat\ list \rightarrow string\hat{*}string$ from positions to formulae. For positions outside the tree, the range of f is defined to be $\check{\in}$, where $\check{\in}$ is shorthand for some dummy expression of the form $string\hat{*}string$. To this end, formal definitions must be presented: f shall be defined to be a well-founded tree; the rule used at each node in the tree prescribes well-formedness; strings shall represent the formula and rule at each node (for formulae correspond to strings); plus the definition of f as a prooftree shall be given. The ultimate goal is to use the notion of effectiveness in order to place a restriction on the ω-rule without the use of numeric encoding. In the following chapter I shall also show what trees my implementational approach selects, and how their correctness may be checked.

5.2.1 Syntax

Next some examples of the sorts of syntax which is involved are presented, and it is shown how this fits into the general framework.

Recall that the syntax used for labelling the tree is composed of variables, constants, terms, formulae and sequents.[1] Terms are the objects of the language, which are generated from constants and variables via functions, and including the former. Formulae are assertions about terms having certain properties, and are generated via predicates and the logical operators. Sequents are composed of a finite (but possibly empty) set of formulae, followed by a turnstile ($\vdash$), followed by at most one formula. Although it may be considered a simplification just to use strings rather than abstract syntax, it is sufficient for the current purposes.

Note that the characters allowable within symbols are $a, \ldots, z, A, \ldots Z, 0, 1, \ldots 9$. Brackets '(' and ')' are used for clarification of grouping. Other symbols involved in forming terms, formulae and sequents are '$+$', '.', ',', '$\vdash$' and the logical connectives. A string is a sequence of these symbols enclosed by ''. Only some of these strings will be meaningful, namely those corresponding to the terms or formulae of the language $PA_{c\omega}$.

The aim is to show that the functions used to label trees with syntax are effective. In fact, recursion over strings is not needed in order to define the string operations below, since these may be built using concatenation ($\hat{*}$), composition and projection plus recursion over lists and numerals. Basic syntax will carry over from PA, such as:

$\mathsf{plus}(X, Y) =_S (X\hat{*}\text{'}+\text{'})\hat{*}Y$;

$\mathsf{times}(X, Y) =_S (X\hat{*}\text{'.'})\hat{*}Y$;

$\mathsf{equals}^2(X, Y) =_S (X\hat{*}\text{'}=\text{'})\hat{*}Y$.

Given the definitions of effectiveness over strings (and hence the syntax) given in Subsection 5.1.3, it may be shown that the basic operations corresponding to the formation of terms in the language, and then the formation of formulae, are also primitive recursive.

plus: $string \times string \rightarrow string$ is effective, where

---

[1] See Section 4.1 and also (Kleene, 1962, P70–80) for syntactic details.

[2] Note that this is the object-level equality.

plus$(Str1, Str2) =_S (Str1\hat{*}\text{`}+\text{'})\hat{*}Str2$.[3]

Similarly, the other term-constructing operations are effective, ie. times: *string* × *string* → *string* where times$(Str1, Str2) =_S (Str1\hat{*}\text{`}.\text{'})\hat{*}Str2$ (via proof by recursion). The operation corresponding to the formation of atomic formulae ie. equals$(X, Y) =_S (X\hat{*}\text{`}=\text{'})\hat{*}Y$ is also shown to be effective in an analogous manner to plus.

Formulae are formed by composition, and by use of logical symbols. So, if X and Y are strings, one may define and: *string* × *string* → *string* such that and$(X, Y) =_S X\hat{*}\text{`}\wedge\text{'}\hat{*}Y$. The definition is given generally, but only used on formulae. Similarly for or and implies: these are all shown to be effective by recursion. not: *string* → *string* defined by not$(X) =_S \text{`}\neg\text{'}\hat{*}X$, also has a recursive definition. In the case of the quantifiers forall/exists: *string* × *string* → *string*, forall$(X, Y) =_S \text{`}\forall\text{'}\hat{*}X\hat{*}\text{`}::\text{'}\hat{*}Y$[4] is effective (and similarly for exists).

Sequents

In order to form sequents, one requires the function turnstile: *string list*×*string* → *string*, and the new symbols "⊢" and ",", such that:

turnstile$([\,], Y) =_S \text{`}\vdash\text{'}\hat{*}Y$;

turnstile$([X], Y) =_S X\hat{*}\text{`}\vdash\text{'}\hat{*}Y$;

turnstile$([H :: T], Y) =_S H\hat{*}\text{`},\text{'}\hat{*}$ turnstile(T, Y).

The notion of parametrised syntax strings is also needed. For example, nth_suc allows the expression of $s^n(Y)$, ie. $\underbrace{s \ldots s}_{n \text{ times}}(Y)$, and is defined by nth_suc: *nat* × *string* → *string*, which is effective by the definition:

nth_suc$(0, Y) =_S Y$;

nth_suc $(s(X), Y) =_S \text{`}s(\text{'}\hat{*}$ nth_suc(X,Y) $\hat{*}\text{`})\text{'}$.

This is closely related to numeral_string: *nat* → *string*, which generates strings of natural numbers '0', '$s(0)$', '$s(s(0))$', etc. and which may be defined by

numeral_string$(n) =$nth_suc$(n, \text{`}0\text{'})$.

---

[3]This applies to more than terms — for example, plus('0', '$s(+$') = '$0 + s(+$'.

[4]According to the Seldon formulation of forall: see Subsection 10.2.

Next the definition of what it is to have a parametrised equation is considered; this is needed to describe the nth iteration of some rewrite rule. For the particular equation $s^n(X) + Y = s^n(X + Y)$, eq: *nat* $\rightarrow$ *string* may be defined by:

$$\mathsf{eq}(n) =_S \mathsf{equals}(\mathsf{plus}(\mathsf{nth\_suc}(n, \text{`}X\text{'}), \text{`}Y\text{'}), \mathsf{nth\_suc}(n, \mathsf{plus}(\text{`}X\text{'}, \text{`}Y\text{'}))).$$

This is an example of the sort of expression found in the lines of the general proof (for they are parametrised — if a particular number is substituted for n, the result will evaluate to the syntax of formulae within sequents associated with individual nodes in the recursive prooftree). Other examples of parametrised syntax are $g(n) =_S$ '$\underbrace{s \ldots s}_{n}$' plus(0,Z) ($=_S$ *nth_suc*(n, plus(0,Z))) and $h(n) =_S$ plus('$\underbrace{s \ldots s(0)}_{n}$', Z)($=_S$ plus (*nth_suc*(n, 0), Z)). The approach extends to all of the syntax of $PA_{c\omega}$, as defined in Section 4.1. Note that, for the general proof, which shall be defined in Chapter 6, it is necessary to show in addition that both the application of rewrite rules is effective, and that the extension of the syntax used within the general proof is effective (see Section 6.3).

5.2.2 Comparison with Effectiveness of nth Subtree of Prooftrees

The effectiveness of syntactical functions f : *nat* $\rightarrow$ *string* such that $f : n \mapsto$ '$\Phi(n)$' is primitive recursive for some formula $\Phi(n)$ compares with the effectiveness of functions f : *nat list* $\rightarrow$ *string* defining proofs. For it is necessary to express the notion of an nth subtree, $s^n(Term)$, in terms of n within the representation of $PA_{c\omega}$, in order to consider the kth subtree (ie. $P(k)$, where $A(\underline{k})$ is the $(k+1)$st hypothesis of the constructive ω-rule $\frac{A(0) \ldots A(\underline{k}) \ldots}{\forall x\, A(x)}$). Hence, there is a correspondence between certain syntactical expressions, such as numeral_string (with domain *nat* and range *string*) and prooftrees. However, as seen in the previous subsection, most of the other syntax is not parametrised in this way.

Now it is possible to tackle expression of the 'primitive recursive' restriction on the ω-rule, in terms of the effectiveness of its subproofs.

5.3 Effective Prooftrees for $PA_{c\omega}$

In the previous section the notion of what it is for a tree to be effective has been defined. It is now possible to move on to provide a definition of prooftrees for $PA_{c\omega}$, part of which is to show that these are effective.

5.3.1 General Approach

Firstly the notion of prooftrees shall be defined. This representation is analogous to giving the semantics of $PA_{c\omega}$, for the well-founded trees are mathematical objects which are the focus of discussion. The semantics of $PA_{c\omega}$ would define what it means to say that a statement X is provable in $PA_{c\omega}$, which in this case would be if there is an effective prooftree which ends up proving X. In Subsection 2.1.2, it was postulated that the meaning of a proof might be represented by a canonical proof; hence the *canonical* effective prooftrees would provide a semantics for $PA_{c\omega}$ (cf. Subsection 5.3.3).

I have defined a tree as *primitive recursive* if there is an effective function which describes it,[5] and I have considered the question of what it is for f to be an effective function $f : nat\ list \rightarrow seq * rule$ in Section 5.2. Note that in general, the aim is to constructively define infinite trees; in particular, I am going to define

---

[5]This leads us to consider what the criteria for equality might be. Will two ω-derivations be the same if the functions which define them are the same, as Sundholm (Sundholm, 1978, P89) suggests? It might be said that it is rather the end result that matters, and the fact that the derivations were generated, and not the similarity of the functions which generated them. As Kreisel writes (Kreisel, 1971, P127):

> "in connection with the topic of criteria for the *identity* of proofs, it is perhaps worth mentioning that we can easily introduce *some* equivalence relation between derivations even if we consider only *extensional* equality (between terms)."

However, the algorithm does matter, as is shown by the existence of input-output functions (eg. Goodstein's function) which are computable by a primitive recursive function, but for which the algorithm may not be represented in a primitive recursive manner, and the similarity of generating functions must form the criteria for the identity of proofs.

a function which determines what is at any individual node in the tree. It is possible to have trees with a non-recursive shape; I wish to exclude such trees from being prooftrees. The prooftrees shall be given by an effective description, and will be labelled with formulae, using effective syntax, and must be correct and well-founded. This makes use of constructive ideas regarding infinite proofs (see Section 2.1).

A derivation in $PA_{c\omega}$ consists of a tree at whose nodes are assigned sequents and rules. An "analysis" of a derivation (here *without* rules) consists in assigning to each node n of the tree one of the expressions $\forall r, \forall l, \ldots, \forall r_\omega, \exists l_\omega, axiom$ in such a way that if $\forall r, \ldots, axiom$ respectively is assigned to the node n, then the sequents directly below the node n have been obtained from the sequent at the node n by the rule $\forall r, \ldots, axiom$ respectively (in the case of *axiom* it is understood that the sequent at n is a lowermost sequent and an arithmetical axiom).

So, the general idea is that infinite $PA_{c\omega}$ derivations may be considered as well-founded trees generated by an effective function from tree positions (finite lists of natural numbers) to the sequent and rule associated with each node, where at each node there is either no branching at all (ie. it is a bottommost node) and an instance of the rule *axiom* is affixed, or there is 1-fold branching (corresponding to the rules $\wedge l$, $\vee r$, $\rightarrow r$, $\exists r$, etc.) or a 2-fold branching (corresponding to the rules $\wedge r$, $\vee l$, $\rightarrow l$, cut, etc.) or an ω-fold branching (corresponding to the rules $\forall r_\omega$ and $\exists l_\omega$). In fact, there will be infinite branching at each node and the nodes which are not needed will be marked with a dummy symbol $\breve{\in}$. These derivations will be effective trees labelled with formulae (ie. each node is labelled with the appropriate sequent to be proved at that stage) and the rule used, which are checked for correct inference.

5.3.2 Definition of Effective Prooftrees for $PA_{c\omega}$

These notions are now formalised:

Definition 5.1 (Effective prooftree for $PA_{c\omega}$) *f is an effective prooftree for $PA_{c\omega}$ if and only if f is an effective function $f : nat\ list \rightarrow seq * rule$ such that f is well-founded and correct.*

Definition 5.2 (Well-founded tree) *f is* well-founded *if it does not have an infinitely deep branch.*

The fact that only finite lists of natural numbers are allowed as positions is built into the definition of *nat list* — what is important here is that every branch terminates. This is encapsulated in 1. below.[6]

Definition 5.3 (Correct tree) *f is* correct *if at any point in the tree, the relationship between each node and its subnodes will be determined by the rule applied.*

This notion is captured in 6. below. The correctness of prooftrees is further discussed in Section 5.4.

I define f: Position of Node $\mapsto$ (Sequent at Node, Rule used at Node), where the range specifies labels, or symbols, in the tree associated with each node, and the position is represented by lists of natural numbers. I will define an ordering on trees in terms of this position representation (where the positions are represented by lists of natural numbers):

Definition 5.4 (Order in tree) *Define the relation $\leq$ on nat list $\times$ nat list by:*

$$Pos1 \leq Pos2 \leftrightarrow \exists l\ Pos2 = Pos1 <> l,\ l \in nat\ list$$

---

[6]So in the representation below, each branch must at some stage have *axiom* or incomplete (see 3. below) associated with a node on that branch (this will actually be the bottommost node, due to the definitions of construction of the tree). The notion of '$\check{\in}$' formulae is retained for empty nodes. See for example (Girard, 1987, P294), for further details of well-founded trees.

This defines $\leq$ as a partial ordering, which will enable the description of the trees as well-founded with regard to this ordering. These definitions hold for the tree being developed top-down. If a node has position $[P_1, \ldots, P_k]$, its subnodes will have positions $[P_1, \ldots, P_k, 0]$, $[P_1, \ldots, P_k, 1]$, etc. The representation would be analogous if the tree were developed bottom-up, as in the representation given by (Löpez-Escobar, 1976).

Definition 5.5 (Empty node) $\check{\in}$, *the empty label, is shorthand for* (dummy seq, dummy rule), *and indicates that there is nothing at a particular node.*

Hence, $f : nat\ list \rightarrow seq * rule$ describes a derivation in $PA_{c\omega}$ for the sequent Φ, where Φ is the sequent at the top of the tree (viz. at the node $[\,]$) if:

1. $\{p : nat\ list | f(p) \neq \check{\in}\}$ is a well-founded tree according to $\leq$.

2. If $f(p) \neq \check{\in}$, then $q_2^1(f(p))$ is a sentence associated with the node p (namely the sequent to be proved), and $q_2^2(f(p))$ is the name of a rule of $PA_{c\omega}$ used to produce its immediate successors, where q is a projection function such that $q_2^1(A, B) = A$ and $q_2^2(A, B) = B$.

3. If p is a bottommost node in the tree, ie. $f(q) = \check{\in}$ for all q such that $p \leq q$, and $f(p) \neq \check{\in}$, then either $q_2^1(f(p))$ is an axiom of $PA_{c\omega}$ and $q_2^2(f(p))$ is *axiom*, or else $f(p)$ is set to incomplete to indicate that the tree is incomplete.

4. If $Pos <> [K]$ $(K \in \mathbb{N})$ is not a bottommost node, then $q_2^1(f(Pos <> [K]))$ is the Kth subgoal of $q_2^2(f(Pos <> [K]))$ applied to $q_2^1(f(Pos))$.

Definition 5.6 (Incomplete tree) *The derivation is incomplete if not all the leaves are closed ie. if* incomplete *is associated with any node in the tree.*

Definition 5.7 (Prooftree) *The derivation will be a prooftree if it is a complete derivation, in other words if all its leaves are axioms (and the others marked as dummy nodes, if appropriate, since there is infinite branching at each node).*

Definition 5.8 (Subgoals) *The* subgoals *of p may be defined as $subgoals(p) = \{q_2^1(f(p <> [n]))|n \in \mathbb{N}, f(p <> [n]) \neq \check{\in}\}$.*

Consequences of this Definition

Properties of the tree will be:

1. Defining br(Rule), where Rule is a rule of $PA_{c\omega}$, as the number of subgoals of Rule (ie. br($\forall r_\omega$)=ω, br($\rightarrow r$)=1, br($\rightarrow l$)=2, etc.), if $L \neq \omega$, where $br(q_2^2(f(Pos))) = L, L \in \mathbb{N}$, then $f(Pos <> [M]) = \check{\in} \ \forall M \geq L, M \in \mathbb{N}$.[7] Since $br(axiom) = 0$, if $p \leq q$ and $p \neq q$ for some position representation p,q, where p is a bottommost node in the tree, then $f(q) = \check{\in}$.

2. If $f(Pos <> [I]) = \check{\in}$, then $f(Pos <> [J]) = \check{\in} \ \forall J \geq I \in \mathbb{N}$.

As mentioned above, Löpez-Escobar (and hence Sundholm (Sundholm, 1978)) develop the tree from the axioms towards the sequent to be proved. Infinitely deep branches are prevented, although there may be infinitely many top nodes. The process may be thought of as a reductive process, as opposed to one of generation. The procedure described above is analogous to this approach, but, in particular, 4., the section which checks that the formula labelling at the nodes corresponds to the proof rules applied, would be replaced by:

4.* If Pos is not a top-node, then the sequents above Pos (ie. $Pos <> [K]$ (where the positioning is defined the other way up) where $K \in \mathbb{N}$, ie. $\{q_2^1(f(Pos <> [K])) : f(Pos <> [K]) \neq \check{\in}\}$) act as premises for an instance of the rule $q_2^2(f(Pos))$, which yields $q_2^1(f(Pos))$ as conclusion.

In this case the rule associated with a node is that used to infer the sentence associated with the node from its immediate predecessors, which may be considered preferable from a theoretical point of view. However, the approach described in this section is more suitable for automation, as it generates the subgoals of the ω-rule, rather than having to check their presence.

---

[7] $\geq$, since the natural numbers, and the subnodes, are taken to start at 0.

5.3.3 Uniform Search

There is a useful discussion to be had concerning uniform search within the prooftrees. If the object is to carry out uniform search, or the ability to generate all possible proofs is considered desirable, it is necessary to input information about which rules to use in a particular proof. If not, a canonical tree could just be generated (for example, by the standard decomposition method (Takeuti, 1987, P44–7)). The way of forming a canonical tree is to operate on the major connective, with some uniform method of decision about the order of choosing the formulae, and any new terms or variables to be used. Infinitary branching of the tree results if the major connective is a universal quantifier on the right of the sequent, or an existential on the left. Such a process may be constrained to satisfy the conditions given in the previous subsection, and thus will describe a method of provision of an effective prooftree corresponding to the proof of a given sequent in $PA_{c\omega}$ (although of course other (non-canonical) prooftrees which also correspond to the proof of such a sequent in $PA_{c\omega}$ are possible). Hence, for each sequent S a possibly infinite tree is defined, from which either a (cut-free) proof of S may be obtained, or an interpretation not satisfying S. Such a canonical prooftree will provide a meaning for the original sequent to be proved: see Subsection 2.1.2 for a general discussion of canonical proofs.

How is it possible to check which prooftrees are well-founded (cf. Definition 5.2), in case the non-canonical prooftrees are also of interest? Although the well-foundedness check is not computable[8], it might be possible to compute all possible rules which are applicable at a particular node. There are only a finite number of rules which may be applied at each node (in fact, $k+1$, where the sequent is $\{A_1, \ldots, A_k\} \vdash C$, since a logical rule may be applied only on the major connective of each formula), although of course there are also thinning and various

[8]The Halting problem reduces to the well-founded tree problem (but not vice versa). The former is unsolvable in the sense that it is not possible to provide an algorithm in terms of n to tell in advance whether a computation will terminate upon input n — see for example (Boolos & Jeffrey, 1980, P28).

other structural rules, plus the possibility of selecting from an infinite but denumerable number of terms when using the quantificational logical rules (and so in this sense there would be an infinite number of possible rules). However, it would be possible to automate search for a proof by constructing some sort of universal search method which would calculate a suitable rule to be applied to *seq*, develop the tree, and then backtrack to the latest failure point in a backwards, depth-first manner. If only one proof were required, the standard decomposition method mentioned above could be used; otherwise, many proofs of a particular sequent might be found using such a uniform search procedure.

This discussion relates to the question of whether the prooftree-generating function should be of the form $f : nat\ list \to seq$, or $f : nat\ list \to seq * rule$. '*rule*' is an enumerated type (although the quantification rules take parameters, thus producing an infinite number of possibilities), and represents each rule used at a tree position (in the form of a list of natural numbers) to derive the corresponding subgoals. This can be inputted via some external source, or else calculated from the form of the sequent, as suggested above. Note that in Löpez-Escobar's analogous representation, the rule is part of the analysis, and not of the tree, although Sundholm uses $seq * rule$ (Sundholm, 1978, P13). So long as it is possible to show that rule application is an effective operation, either option is possible within the framework given above.

In conclusion, I shall retain the labelling notation of $seq*rule$, since both a sequent and a rule are associated with the node. The rule may be used in order to establish correctness.

5.3.4 Summary

In this section a characterisation of primitive recursive prooftrees has been given. In the next chapter another description of ω-prooftrees is given, closer to practical interests. I have described infinite prooftrees in some metatheory in a finite way, and restricted attention to constructive ones. Effectiveness has been defined over various datatypes, and the constructive trees to be considered are the effective

trees. The next section considers how it would be possible to prove that these trees are correct.

5.4 Correctness of Well-Founded Trees

There are two main notions of correctness of well-founded trees, namely 'local' correctness, which checks that an appropriate rule is applied at each node of the tree, and 'global' correctness, which is concerned with whether the tree is well-founded. Such a distinction is highlighted by Kreisel (Kreisel, 1971, P128):

> "In the case of, possibly, *infinite* derivations there is the additional step of verifying that each (infinite) proof figure involved is well-founded. More explicitly, we give a *description* of a sequence of proof figures ... What has to be established is first that the figure described is *locally* correct, that is the formula at a node N of a proof figure is built up according to the rules from the formulae at the immediate successor of N; second in the infinite case, that the whole figure is well-founded;"

Given a tree in the form of $f : nat\ list \rightarrow string$, how is it possible to check that this is a locally and globally correct tree? In the previous sections well-founded prooftrees have been discussed, and hence global correctness, but the question of local correctness of prooftrees was not properly considered. To check for local correctness, meta-induction is used over the prooftrees. In order to define this, induction principles over trees, lists, strings, etc. are used. There is a natural way of defining induction corresponding to effectiveness (using the base case, constructors, etc.) In general, for the datatypes *nat*, *type list* and *string* given above, the induction rule for each type is of the form:

$$\frac{A(b) \quad A(\vec{r}) \Rightarrow A(c(\vec{r}))}{\forall x \in type\ A(x)}$$

where b is the base case of the datatype *type*, and $c(\vec{r})$ is the constructor of *type* applied to $\vec{r}$ (arbitrary arguments of *type*).

The base cases and constructors for the various types under consideration are given in the previous sections. So, for example, induction over lists will take the

form:

$$\frac{A([\,]) \quad A(t) \Rightarrow A(h :: t)}{\forall l \in type\ list\ A(l)}$$

for arbitrary $t \in type\ list$ and $h \in type$.

To show local correctness of the prooftrees, the corresponding meta-induction (for *nat list*) is used for the tree-defining function $f : nat\ list \rightarrow string * string$. Thus:

$$\frac{f([\,]) \quad f(Pos) \Rightarrow f(Pos <> [k])}{\forall x\ f(x)}$$

where $Pos, x \in nat\ list$ and $k \in nat$. That is to say that given an initial sequent (and initial rule used), plus a way of obtaining from a sequent at some position the sequent at a node directly below that position, then the sequent at each node of the tree is defined. This process of obtaining sequents at subnodes, given a sequent at a node, is carried out by applying the rule associated with the node to the sequent at the node, and is described above. The result is uniquely determined, given a rule and sequent, and hence the prooftrees are locally correct. (This formulation of induction for prooftrees corresponds to meta-induction in $PA_{c\omega}$, which is discussed in Subsection 7.2.2, for which a well-founded tree will be shown to be correct by proving that the sequent at $Pos <> [k]$ may be reduced in a uniform manner to the sequent at Pos.) Note that the induction over the partial ordering $\leq$ of the tree representation is usually given as transfinite induction, which is allowable if numeric encoding is used at each node (Kreisel, 1965, P163).

At this stage it could be objected that there might be circularity, for although the ω-rule is used instead of induction, meta-induction is being introduced here, which might result in there being no advance. This question will be addressed in Subsection 8.6.2, in which it shall be shown that the generalisation problem does not in practice occur in the theory of trees, and so that there is a gain after all.

5.5 Overall Conclusions

In this chapter a characterisation of $PA_{c\omega}$ has been presented. I have chosen to use the primitive recursive ω-rule, rather than its recursive counterpart. When reasoning about the enlarged system, the objects of interest are recursive (possibly infinite) prooftrees (in the sense of Löpez-Escobar (Löpez-Escobar, 1976)), labelled with formulae (namely, the sequents to be proved at each point) and rules. The notion of effectiveness of a tree, which corresponds to primitive recursion, has been defined. A (proof) tree must be well-founded, in the sense that it does not have an infinitely deep branch. The rules that relate the formulae between node and subnode are the standard rules for the logical connectives, the extra ω-rule with subgoals $\Phi(\underline{0}), \Phi(\underline{1}), \ldots$, and substitution. A formula in PA is demonstrated in the extended theory by exhibiting a prooftree labelled at the root with the given formula. In the next chapter I shall show how the implementation relates to the effective prooftrees, and how properties of the trees are verifiable.

Chapter 6

General Proofs

In this chapter the relationship between $PA_{c\omega}$ and the implementational version of such a system is considered. More specifically, the following are examined:

- presentation of the general proof representation (involving the application of rewrite rules);
- discussion of the implementational approach;
- consideration of the relationship between the general proof representation and $PA_{c\omega}$;
- correctness of the general proof representation;
- discussion of conditional rewrite rules and branching general proofs.

6.1 General Proof Representation

The general proof representation used in the implementation for representing and manipulating proofs shall be presented. In the following sections the relationship between this representation and the system $PA_{c\omega}$ shall be detailed, and then various properties of the general proof representation investigated.

6.1.1 What is a General Proof?

As discussed in Chapter 4, one way in which the constructive ω-rule may be put into effect is to require that there is an enumeration of the derivations which prove the premises — for example one could code proofs by numbers, by means of a primitive recursive function which generates them, as described in Section 2.4. But I have not used such a traditional representation; it was sufficient for my purposes to provide (for the nth case) a description for the general proof in a constructive way.

The general proof representation represents $P(n)$, the proof of the nth numerator of the constructive ω-rule, in terms of rewrite rules applied a function of n or a constant number of times to formulae (dependent upon the parameter n). There are two types of representation of the general proof which will be considered: an implementational representation and the representation in terms of trees described in Chapter 5, which are closely related. It is necessary to provide (for the nth case) a description for the general proof in a constructive way (in this case a recursive way), which captures the notion that each $P(n)$ is being proved in a uniform way (from parameter n). The method of obtaining a general proof of a statement $A(r)$ is to take the nth case $A(\underline{n})$, and apply the rewrite rules indicated in the hope of reaching truth or equality. For example, let us consider $\forall x\ (x+x)+x = x+(x+x)$. Let us presume that, within the particular formalisation of arithmetic chosen, one is given the axioms of addition of Figure 6–1. The general proof, $P(n)$, will take the form given in Figure 6–1 (although it may be represented in a variety of ways).

By $s^n(0)$ is meant the numeral $\underline{n}$, ie. the term formed by applying the successor function n times to 0. The next stages use the axioms as rewrite rules from left to right, and substitution in the general proof, under the appropriate instantiation of variables, with the aim of reducing both sides of the equation to the same formula.

The general proof represents, and highlights, blocks of rewrite rules which are being applied. Meta-induction may be used (on the first argument) to prove the more general rewrite rules from one block to the next: for example,

$$\forall n\ s^n(x) + y = s^n(x + y)$$

Axioms

$$0 + y = y \tag{6.1}$$
$$s(x) + y = s(x + y) \tag{6.2}$$

Proof

$\underline{n} \equiv s^n(0)$
USE 6.2 n *TIMES ON RIGHT*
USE 6.2 n *TIMES ON LEFT*
USE 6.1 *TWICE*
USE 6.2 n *TIMES ON LEFT*

$$(\underline{n} + \underline{n}) + \underline{n} = \underline{n} + (\underline{n} + \underline{n})$$
$$(s^n(0) + s^n(0)) + s^n(0) = s^n(0) + (s^n(0) + s^n(0))$$
$$(s^n(0) + s^n(0)) + s^n(0) = s^n(0 + (s^n(0) + s^n(0)))$$
$$s^n(0 + s^n(0)) + s^n(0) = s^n(0 + (s^n(0) + s^n(0)))$$
$$s^n(s^n(0)) + s^n(0) = s^n(s^n(0) + s^n(0))$$
$$s^n(\underbrace{s^n(0)}_{A} + \underbrace{s^n(0)}_{B}) = s^n(\underbrace{s^n(0)}_{C} + \underbrace{s^n(0)}_{D})$$
$$EQUALITY$$

Figure 6–1: A General Proof of $\forall x\ (x + x) + x = x + (x + x)$

corresponds to n applications of axiom 6.2 above.

Proof

Base: $n = 1$ $s(x) + y = s(x + y)$ — 6.2.

Step: If $s^r(x) + y = s^r(x + y)$, then $s^{r+1}(x) + y = s^{r+1}(x + y)$.

$$\begin{aligned} s^{r+1}(x) + y &= s.s^r(x) + y \\ &= s(s^r(x) + y)\ by\ 6.2 \\ &= s.s^r(x + y)\ by\ hyp \\ &= s^{r+1}(x + y)\ \square \end{aligned}$$

This type of meta-induction used to justify the correctness of the general proofs shall be further considered in Subsection 6.4.1.

The recursive function which is sought is described by the sequence of rule applications, parametrised over n. The general proof is given in terms of n, each rule R being applied $f_R(n)$ times, for some f_R. In practice, the first few proofs will be special cases, and it is rather the correspondence between the proofs of $P(99)$, say, and $P(100)$, which is to be captured. The question of whether the goal is just being transferred to the meta-level in the general proof, where pre-defined rules are used to obtain equality, thus obtaining a form of circularity, is addressed in Subsection 8.6.2.

6.1.2 Implementational Approach

The above discussion results in the consideration of how the general proofs might be represented from an implementational point of view. For an application of the ω-rule, a description of the nth subproof is required, uniformly with respect to the parameter n. An example representation, in the linear case, for the general proof for $P(n)$ is:

$$gen([R1(Pos1, f1'(n)), R2(Pos2, f2'(n)), \ldots]).$$

where the representation for each individual proof for $P(J)$ would be of the form:

$$proof(J, [Rule1_J(Pos1_J, T1_J), Rule2_J(Pos1_J, T2_J), \ldots])$$

This is a predicate, its first argument being the natural number J, and the other argument being an ordered list of predicates such that $RuleK_J$ is the Kth rule to be applied in the proof, and is applied TK_J times, with $PosK_J$ specifying the position at which it is applied. Note that one may have $RuleA_J = RuleB_J$. For further details see Section 10.3. Note that in general a more complicated representation is needed (see Section 6.6), since it is necessary to know exactly where to apply the rules in the proof, particularly if branching occurs, and perhaps the best way of pinpointing this information is to use a tree representation.

An analogous representation in terms of prooftrees may be used. A recursive function $gen\_pf : proof \times \mathbb{N} \rightarrow proof$ is defined which, when given a number and a (proof)tree, returns the prooftree below some node in the tree whose position is a function of that number. So if the ω-rule is used initially (at node $[\,]$), the $n+1$th subtree of the tree may be selected by using $gen\_pf(Tree, n)$, which will be the prooftree with top node $[n]$ (for $0 \leq n \in \omega$). This enables the construction of the generalised proof, in the sense that it will represent the mechanism of obtaining the proofs $P(0), P(\underline{1}), P(\underline{2}) \ldots$ generated by the constructive ω-rule. It can be seen that this is closely related to the function which generates recursive prooftrees given in the previous chapter.

6.2 Relationship Between General Proof Representation and $PA_{c\omega}$

In this section the relationship between the representation of proofs given in Chapter 5 and those given in Section 6.1 above is defined.

6.2.1 The Derived Rule $\mathcal{I}$

In the representation of $PA_{c\omega}$, a derived form of primitive recursive function-defining equations as inference rules is required to allow rewriting of formulae (which takes place in general proof examples, and hence an analogy is needed). These are of the type $\mathcal{I}$ below — it is just convenient to define these as single steps, as shorthand in order to avoid going through all the equivalent tedious steps of PA.

Axiom: $\forall x \forall y\ s(x) + y = s(x + y)$

Example General Proof: $\dfrac{s(0)+0=s(0)}{s(0+0)=s(0)}\mathcal{I}$

In general:

$$\frac{\Gamma \vdash \forall xy R(x,y) = S(x,y) \quad \Delta \vdash T(R(p,q))}{\Gamma,\ \Delta \vdash T(S(p,q))}\mathcal{I}$$

$\mathcal{I}$ is a derived rule of PA, as it is equivalent to the following:

$$\dfrac{\dfrac{\dfrac{}{R(p,q) = S(p,q) \vdash R(p,q) = S(p,q)}\,axiom}{\forall xy\ R(x,y) = S(x,y) \vdash R(p,q) = S(p,q)}\,\forall l(\times 2) \qquad \Gamma \vdash \forall xy\ R(x,y) = S(x,y)}{\Gamma \vdash R(p,q) = S(p,q)}\,cut \qquad \Delta \vdash T(R(p,q))$$

$$\Gamma,\ \Delta \vdash T(S(p,q))\ subst$$

One may deduce a more general rule $\mathcal{I}'$, in a similar manner, where $\mathcal{I}'$ is the following:

$$\frac{\Gamma \vdash \forall \vec{x} R(\vec{x}) = S(\vec{x}) \quad \Delta \vdash T(R(\vec{p}))}{\Gamma,\ \Delta \vdash T(S(\vec{p}))}\mathcal{I}'$$

where if the arity of $\vec{x}$ is k, then $\vec{p} = \{p_1, \ldots p_k\}$ for terms p_i $(1 \leq i \leq k)$ in the language of $PA_{c\omega}$.

The derived rule $\mathcal{I}$ above corresponds to "apply $\mathcal{R}$ once", where $\mathcal{R}$ is the rewrite rule $R(x,y) \Rightarrow S(x,y)$. "Apply $\mathcal{R}$ a constant number of times, k" just repeats $\mathcal{I}$ k times. $\mathcal{I}$ corresponds to the basic formulation of $PA_{c\omega}$, so if the given procedure about tree construction from rules is followed, a recursive prooftree will be obtained. Note that in the tree there is infinite branching, so dummy subtrees will have to be inserted if necessary.

In the general proof, one starts with something to be proved, and ends with equality, corresponding to an axiom. In practice, only rewrite rules are usually used, but it is possible to use logical rules which might cause the general proof to split. However, such case splits of conditionals do not pose any problem, because they correspond exactly to the logical rules of $PA_{c\omega}$.

Note that if such derived rules were to be used in the prooftree representation, the whole equivalent subtree should be substituted instead, since the new node position otherwise would not be a direct subnode (in terms of its position representation) of the original node at which such a rule was applied.

6.2.2 The Derived Rule $\mathcal{J}$

The other case to consider is when a rewrite rule of the form "apply $\mathcal{R}$ some function of n times" is applied in the general proof. This corresponds to the (derived) rule $\mathcal{J}$ below, which may be proven in $PA_{c\omega}$, with the use of induction.

$$\frac{\Gamma \vdash \forall x \forall y R(x,y) = S(x,y) \quad \Delta \vdash T(R^{f(n)}(p,q))}{\Gamma, \Delta \vdash T(S^{f(n)}(p,q))} \mathcal{J}$$

where $S^{f(n)}(p,q)$ is the result of applying the rule $\mathcal{R}$ $f(n)$ times to $R^{f(n)}(p,q)$ (assuming that such a rule application may be carried out). One such example is that considered in Subsection 6.1.1, in which $s^n(x) + y \Rightarrow s^n(x+y)$ corresponds to n applications of the rewrite rule $s(x) + y \Rightarrow s(x+y)$. It is also possible to derive a more general version $\mathcal{J}'$, which is analogous to $\mathcal{I}'$.

The proof in $PA_{c\omega}$ of $\mathcal{J}$ is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{R(R^r(p,q))=S(R^r(p,q))\vdash R(R^r(p,q))=S(R^r(p,q))}\,ax}{\vdash R(R^r(p,q))=S(R^r(p,q))}\,\forall l \qquad \Gamma\vdash \forall xyR(x,y)=S(x,y)}{\Gamma\vdash R(R^r(p,q))=S(R^r(p,q))}\,cut \qquad \dfrac{}{T(S(R^r(p,q)))\vdash T(S(R^r(p,q)))}\,ax}{\vdash T(S(R^r(p,q)))\to T(S(R^r(p,q)))}\,\mathcal{I}}{T(R^r(p,q))\to T(S^r(p,q))\vdash T(R^{s(r)}(p,q))\to T(S^{s(r)}(p,q))}\,\mathcal{I} \quad \star}{\Gamma\vdash T(R^n(p,q))\to T(S^n(p,q))}\,ind}{\Gamma, T(R^n(p,q))\vdash T(S^n(p,q))}\,\to r \qquad \Delta\vdash T(R^n(p,q))}{\Gamma,\Delta\vdash T(S^n(p,q))}\,cut$$

At $\star$ there is the prooftree:

$$\dfrac{\dfrac{\dfrac{}{R(p,q)=S(p,q),T(R(p,q))\vdash T(S(p,q))}\,ax \qquad \dfrac{\dfrac{}{\Gamma\vdash \forall xyR(x,y)=S(x,y)}\,ax \qquad \dfrac{\dfrac{}{R(p,q)=S(p,q)\vdash R(p,q)=S(p,q)}\,ax}{\forall xyR(x,y)=S(x,y)\vdash R(p,q)=S(p,q)}\,\forall l}{\Gamma\vdash R(p,q)=S(p,q)}\,cut}{\Gamma\vdash T(R(p,q))\to T(S(p,q))}\,cut$$

The justification for use of induction in the proof is that induction is a derived rule in $PA_{c\omega}$ (see Chapter 7). At this stage the question might be raised that this all seems rather circular, for although the ω-rule is being considered instead of induction, induction in $PA_{c\omega}$ is being used, which might leave us no better off. This question will be discussed in Subsection 8.6.2.

A summary of the above consideration of the relationship of the general proof representation with $PA_{c\omega}$ is that the proof of $A(\underline{n})$ in the ω-rule $\frac{A(0),\ldots A(\underline{n})\ldots}{\forall xA(x)}$ is represented in the general proof using notation which is not that of $PA_{c\omega}$, but which is more suitable for implementation. However, the notation (represented by derived rules $\mathcal{I}$ and $\mathcal{J}$), may be converted to that of $PA_{c\omega}$, as may be seen by the fact that the rules $\mathcal{I}$ and $\mathcal{J}$ correspond to proofs in $PA_{c\omega}$.

So, the general proof can be accounted for in terms of $PA_{c\omega}$.

6.3 Proving Properties: Syntax of General Proof Representation

Description of syntax shall be used to indicate the system in which correctness of the general proof system is proved (that is, showing that anything provable in the general proof representation will be provable in the recursive prooftree representation, ie. $PA_{c\omega}$). In doing so, use is made of inductive principles over the syntax.

Syntax parametrised by n has already been shown to be 'effective' in Subsection 5.2.1. In this section I wish to give a primitive recursive description of the operations on the syntax in the general proof representation, which will involve giving a primitive recursive description of what it is to apply a rewrite rule a function of n times, or a constant number of times. So the aim is to define, primitive recursively, the following (cf. *apply* on page 118):

`rewrite(1,String1,String2)` if ...

`rewrite(s(X),String1,String2)` if ...

$\forall n$ `rewrite(n,g(n),h(n))` if ...

where `g,h`: $nat \rightarrow string$, as defined above.

I define first of all a basic function with strings and instantiation. Suppose the appropriate rewrite rule is $R : R1 \Rightarrow R2$.

`rewrite(1,R,E1,E2)` if (`get_rule(R,R1,R2)` and `match(E1,R1,Z)` and `match(E2,R2,Z)`).

`rewrite(s(X),R,E1,E2)` if (`rewrite(1,R,E1,E3)` and `rewrite(X,R,E3,E2)`).

where `match`$(String, Formula, Vars)$ matches $String$ (rewritten as a formula) with $Formula$, with unification variables $Vars$, and `get_rule`$(R, R1, R2)$ returns the appropriate left and right sides ($R1$ and $R2$ respectively) of the rewrite rule R. Note the connection between the first procedure (of a single application of R) and the use of the derived rule $\mathcal{I}$, which is defined in Subsection 6.2.1: the latter rule would essentially carry out this stage. Now I have to justify using a rewrite rule n times (which is done to give the general proof). The justification will involve

induction over the syntax. The induction rule is of the form:

$$\frac{\vdash g(0) \rightarrow h(0) \quad g(r) \rightarrow h(r) \vdash g(s(r)) \rightarrow h(s(r))}{\vdash \forall n\ g(n) \rightarrow h(n)} ind \quad *$$

One could prove $*$ by rewriting $g(s(r))$ to $j(g(r))$ and $h(s(r))$ to $j(g(r))$.

Another, more formal, approach is to define: $\mathsf{rewrite}(n, g(n), h(n)) =_S \mathsf{eq}(n)$, where $eq(n)$ is as given above on page 97, and is essentially $\mathsf{equals}(g(n), h(n))$. Then the idea is to show that $\forall n.\ \mathsf{eq}(n)$ is a theorem of $PA_{c\omega}$. Note that for each inference rule in the object logic, there is a rule (corresponding to substitution),

$\vdash \mathsf{theorem}\ (\mathsf{equals}(A, B)) \rightarrow \mathsf{theorem}(\Phi(A)) \rightarrow \mathsf{theorem}(\Phi(B))$,

where Φ is any formula in which A may appear.

There is also, to exclude X being ')',

$\vdash \mathsf{term}(X) \rightarrow \mathsf{theorem}(\mathsf{equal}(X, X))$.

I will work through an example to illustrate the procedure. By induction:

Base case : $\vdash$ '$s(X) + Y = s(X + Y)$' $-$ *axiom.*

Step case : *suppose* $\vdash$ '$s^n(X) + Y = s^n(X + Y)$'.

To show : $\vdash$ '$s^{s(n)}(X) + Y = s^{s(n)}(X + Y)$'.

ie. $\vdash$ theorem(equals(plus(nth_suc($s(n)$,'X'),'Y'),nth_suc($s(n)$, plus('X','Y'))

ie. $\vdash$ theorem(equals(plus('$s($' $*$ nth_suc(n,'X') $*$ '$)$','Y'),...)

Now the induction hypothesis should be used. So:

$\vdash$ theorem(equals(plus(nth_suc($s(n)$,'X'),'Y'),'$s($' $*$ nth_suc(n, plus('X','Y') $*$ '$)$')),...)

(*subgoal*) $\vdash$ theorem(equals(plus(nth_suc($s(n)$,'X'),'Y'),'$s($'$*$plus(nth_suc(n,'X'),'Y') $*$ '$)$'))

This proves that, for all n, $\vdash s^n(X) + Y = s^n(X + Y)$ is a theorem, and hence that the rewrite rule from $s^n(X) + Y$ to $s^n(X + Y)$ is justified, assuming the rewrite rule $s(P) + Q \Rightarrow s(P + Q)$ at the object level (the former is just this rule applied n times).

In conclusion, I have shown which operations I consider to be primitive recursive on the syntax of the general proof representation, and these include the operations of the application of rewrite rules.

6.4 Correctness of the General Proof Representation

I shall now check further properties of the general proof representation. In Section 5.4 correctness of well-founded trees was discussed, and a distinction made between local and global correctness. In this section I shall demonstrate that the implementational approach is globally correct, in the sense that it is sound with respect to the presentation of $PA_{c\omega}$ given in Chapter 5, and also (in Subsection 6.4.1) that it is locally correct, in that the rewrite rule applications specified correspond at each node to a general proof representation.

Soundness entails that only true statements of $PA_{c\omega}$ are provable via the general proof representation. So for this case it is necessary to show that the resulting formulae from the application of rewrite rules correspond to the application of proof rules in $PA_{c\omega}$. I shall show that for each general proof construct, there is a corresponding primitive recursive prooftree of $PA_{c\omega}$. In other words, it is necessary to show what the effect of the rewrites is, and check that the formula labelling at the nodes corresponds to the application of proof rules in $PA_{c\omega}$. So it must be shown that "rule applied once/constant/$f(n)$ times" in the general proof corresponds to (in the sense of giving) correct subtrees in the prooftree. Now, the general proof is only dealing with a branching (ω) point, and the subtree below that. The general proof corresponds to a subtree of the ω-rule. It is merely necessary to show that the representation used in the implementation is a derived form of the structures given in the previous chapter. The essential task of showing that the application of rewrite rules may be represented in $PA_{c\omega}$ has been carried out in Section 6.2 above.

Hence, global correctness necessitates showing that for each general proof construct there is a primitive recursive function which indicates what is at any particular node in the tree representation described above. This is possible by inspecting the tree diagrams for the derived inference rules $\mathcal{I}$ and $\mathcal{J}$ above. The primitive recursive function would be analogous to the definition of f in the previous chapter.

When "apply rule $\mathcal{R}$ k times" appears in the general proof, the function would generate the tree as previously described, but using the rules from $\mathcal{I}$ (repeated as appropriate), or $\mathcal{J}$, and generating the rest of each infinite layer as dummy variables. Of course, the layer is not literally filled in, as this will be a non-terminating process. It is merely necessary to note that, for example, $f(Pos <> [l]) = \check{\in} \forall l \in nat\ list$, and that any particular individual position could be checked as yielding $\check{\in}$. This is enough to give correctness of the tree. A tree with infinite branching points may be generated using the constructive ω-rule (ie. from the general proof). This should be using a depth-first generation, because the tree is required to be well-founded; in this way the tree would in essence be completed — after a certain point with generating the subgoals of the ω-rule, the case for the kth subtree could be given if termination was required.

The function that generates the general proof has properties of f (which is a primitive recursive prooftree for $PA_{c\omega}$). When implementing global correctness of the general proof, one is proving properties of functions from natural numbers to strings (the application of rewrite rules a function of n times, where the (basic) rewrite rule is a function from strings to strings). I will want to prove properties of the rewrite-rule application $apply : rewrite\ rule * tree\_pos * nat * string \rightarrow string$. (The third argument is the number of times applied, the fourth is the initial expression, and the range is the final expression.) So, for example, there is $apply(rewrite1, P, n, t_1(n)) = t_2(n)$, where t_1, t_2 are of the syntactic form described in the previous section. Meta-induction is used to justify such statements, where the rewrite rule is applied a function of n times, as shown in the following section.

When the constructive ω-rule is used, one would wish to show that, given the representation $f : n \rightsquigarrow$ '$\Phi(n)$' discussed in Subsection 5.2.2, the n in the syntax denoted by $\Phi(n)$ is primitive recursive, in order to show that the primitive recursive restriction on the ω-rule does indeed apply. Such recursiveness has been shown above in Section 6.3.

The general proof $P(\underline{k})$ will represent the kth subtree below use of the ω-rule on some sequent of the form $\forall x \Phi(x)$. Hence, $P : nat\ list \rightarrow seq * rule$ describes a subtree. The sequents in the nth subtree (ie. the general proof) are defined by:

$P(Pos :: n <> l) = Seq * Rule$ for some $l \in nat\ list$, where $q_2^2(f(Pos))$ is the constructive ω-rule. P is f, the generating function for the whole tree, but is just defined over this domain. This defines a primitive recursive P which represents the general proof. However, this is not the way that P is defined in the implementation, and hence it is necessary to show the correspondence between them. It is necessary to show that the notion of a rewrite rule being applied n times is both primitive recursive (which must itself be defined over this domain), and sound with respect to the theory of $PA_{c\omega}$. For the general proof, one could define the representation $genpf : rule*subpos*nat*tree\_pos*seq \rightarrow seq$, which is analogous to the function *apply* mentioned above. Primitive recursion is demonstrated by the operations of adding a single branch to extend the tree, and induction to prove the nth case, as shown in the following subsection.

In conclusion, in order to convert from general proofs to a recursive prooftree form, it is necessary to substitute k for $\underline{n}$, where $k \in \mathbb{N}$, to get the kth subtree. Rewrite rules should be converted to the appropriate rules of $PA_{c\omega}$ (from the trees given in Subsections 6.2.1 and 6.2.2 for $\mathcal{I}$ and $\mathcal{J}$) and applied as appropriate; this also covers the case of the application of rules $f(n)$ times.

6.4.1 Induction over General Proofs

The correctness of well-founded prooftrees has been discussed above in Section 5.4, and global correctness of the general proof representation has been discussed above. I now move on to consider local correctness. Corresponding to the general representation there is an effective prooftree; I shall consider how this can be verified (ie. shown to be correct at each node).

Within the general proofs use is made of generalised forms of derived rules of inference, representing application of these rewrite rules n times, such as $s^n(x) + y \Rightarrow s^n(x+y)$ from $s(x)+y \Rightarrow s(x+y)$. These generalised forms may be proved by meta-induction on n, as shown in Subsection 6.1.1 (note that meta-induction over prooftrees will be further considered in Subsection 7.2.2). Such meta-induction is

carried out in the theory of parametrised syntax considered in the previous section, and in Subsection 5.2.1.

Of relevance also is the discussion given in Subsection 10.3.1, which presents an implementational method for checking for correctness of the general proof, in the sense that the general proof gives an ω-proof[1] of the statements which it claims to be proving (that is, that it is the right answer to the inductive inference problem). Thus, the implementational section on correctness given in Subsection 10.3.1 attempts to prove such generalised statements, ie. to justify such (derived) rewrite rules used in the general proof by means of such meta-induction.

By justifying the general proof, the corresponding effective prooftree is also justified. The overall process is that the correctness of each branch is being checked at the meta-level: given various initial formulae with a common structure, a parametrised representation of these formulae is constructed via inductive inference (which corresponds to the initial line in the general proof) and the derived (parametrised) rewrite rule is applied to give a new parametrised representation (corresponding to the new line of the general proof), from which the new individual formulae may be derived.

In general, there are two sorts of operations to extend the prooftree: applying a rewrite rule R at some tree position $Pos(n)$ will give an updated tree with a single node added (type 1), and applying a rewrite rule R some function of n times at a given position will return a new tree with that number of extra nodes (type 2). These may then be tacked together to form the final general proof. (Meta)induction may be used to justify type 2 applications. Thus, given $gen\_pf : proof \times \mathbb{N} \rightarrow proof$, as described on page 111, if one wishes to prove $\forall n \Phi(gen\_pf(Tree, n))$, induction will require proof of $\Phi(gen\_pf(Tree, 0))$ and, given a proof of $\Phi(gen\_pf(Tree, x))$, to show that there is a proof of $\Phi(gen\_pf(Tree, s(x)))$, where x is any natural number.

---

[1]Note that an ω-proof is a proof in $PA_{c\omega}$ with initial rule used being the ω-rule: hence the general proof is a parametrised version of an arbitrary subtree of this proof, using certain derived rules.

In summary, it is possible to extend the tree by a rule applied a constant number of times, which may be calculated straightforwardly via iteration of a single extension, or else by a function of a natural number, $f(n)$ say, times, which will be justified by induction on n (over the inductive structure), plus quantification over the object-level syntax. More specifically, this justification will take the following form. Suppose a rewrite rule R is applied n times at position $Pos(n)$, where $R : R1 \Rightarrow R2$. Let K^n be defined as the result of applying R n times to K. The goal is to prove that $\forall k\ \Phi(R1^k) \Rightarrow \Phi(R2^k)$. The base case of induction is to prove $\Phi(R1) \Rightarrow \Phi(R2)$, which is provable by comparison with rule R. The step case assumes that $\Phi(R1^n) \Rightarrow \Phi(R2^n)$ and wishes to prove $\Phi(R1^{n+1}) \Rightarrow \Phi(R2^{n+1})$. There is:

$$\begin{aligned}
\Phi(R1^{n+1}) &= \Phi(R1 * (R1^n)) \\
&\Rightarrow \Phi(R1 * (R2^n)) \quad \textit{by hyp} \\
&\Rightarrow \Phi(R2 * (R2^n)) \quad \textit{by } R \\
&= \Phi(R2^{n+1})
\end{aligned}$$

where $A * B$ means "apply rule A, then apply rule B". So, given $R1, R2 : Seq \rightarrow Seq$, I define $R1 * R2(Seq) \equiv R2(R1(Seq))$.
This process may be justified by the following (condensed) proof of $PA_{c\omega}$:

$$\dfrac{\dfrac{}{R1 = R2 \vdash f(R1) = f(R2)}\,ax \qquad \dfrac{\dfrac{\dfrac{}{f(R1^n) = f(R2^n) \vdash f(R1^n) = f(R2^n)}\,ax}{\dots \vdash f(R1 * (R1^n)) = f(R2 * (R2^n))}\,subst}{R1 = R2,\ f(R1^n) = f(R2^n) \vdash f(R1^{n+1}) = f(R2^{n+1})}\,def}{R1 = R2 \vdash f(R1^k) = f(R2^k)}\,ind$$

Note that an individual case, namely of proving the new rule $\forall n\ s^n(x)+y \Rightarrow s^n(x+y)$, has already been considered in Subsection 6.1.1, and that Subsection 6.2.2 has justified the application of such rules in the general proof, in terms of being a derived rule of $PA_{c\omega}$. Subsection 10.4.1 outlines the implementational algorithm used to justify this process.

In conclusion, the general proof specifies a particular mathematical object, which is a primitive recursive function of the type described in Chapter 5, and local correctness of the general proof may be demonstrated using meta-induction.

6.5 Completeness of General Proof Representation

I now turn to the question of completeness, namely whether the general proof representation is strong enough to represent any (primitive) recursive tree of $PA_{c\omega}$ More specifically, given an arbitrary primitive recursive tree that satisfies the conditions 1.–4. on page 101 (ie. a prooftree of $PA_{c\omega}$), is there always a corresponding general proof notation (involving the notion of applying rewrite rules)? Note that one may carry out meta-induction over the prooftrees[2] (which is used to justify the general proof). The original prooftree generated will be of the form (with recursion): (a) Seq to Seq1 (and possibly Seq2) by a rule of $PA_{c\omega}$, plus infinite dummy nodes, or (b) Seq to an infinite number of sequents generated by an ω-rule ($\forall r_\omega$ or $\exists l_\omega$). One method of attempting to show completeness would be to use meta-induction over prooftrees: first the base case (of a basic prooftree being covered by a general proof) would be considered, and then, assuming that a given prooftree is covered by the general proof notation, one would try to show that that the prooftree extended by the application of a rule at a node is also covered by the general proof notation. An extension of type (a) is allowed for in the general proof notation for case-splits and branching: the classical form of the rewrite rules suggests the appropriate case split on the general proof, so that $\vdash A \wedge B$ just gives A, B in the general proof but $\vdash A \vee B$ causes a case split, for example. An extension of type (b) corresponds to use of the ω-rule at a node in the prooftree. The subsequents may be represented by the new general proof line $A(n, \underline{m})$ from the previous general proof line $\forall x A(x, \underline{m})$ (where the former was originally derived from $\forall y \ldots \forall x A(x, y)$). The new general proof refers to the nth subgoal of the prooftree: substituting k for n in A, where $0 \leq k \leq \omega$, corresponds to the subnodes generated within the prooftrees. In general, the relationship of the general proof representation with that of the effective prooftree representation described

[2]See Chapters 5 and 7.

in the previous chapter is that the general proof corresponds to a generalisation of the nth subtree below use of $\exists l_\omega$ or $\forall r_\omega$ in a prooftree derivation. However, this argument is neither convincing nor rigorous.

Indeed, it does seem as though what is allowed in the hypotheses of the ω-rule by the general proof representation, namely parametrisation over the number of applications of a rewrite rule, is not general enough (this ties in with the discussion in the following section about whether branching rules in the general proof are directly analogous to the use of logical rules in an ω-proof). It is the case that, as shown for example in Subsection 6.2.2, rewrite rules may be used in a substitution into a general predicate (of arbitrary logical complexity), and hence the general proof representation (which represents the proof of the nth numerator of the constructive ω-rule in terms of the application of rewrite rules to formulae) does not imply that all proofs considered are of equations. However, the general proof representation does limit what happens in an ω-rule application to what can be achieved by iterated substitution. Hence, it would be better just to concentrate on the soundness aspect, since completeness is not actually provable.

6.6 Branching General Proofs and Conditionals

In this section the question of branching general proofs is introduced, and it is considered whether branching within the general proof is really necessary, and if so, how this might be represented.

6.6.1 Branching Rules

Branching rules are rules which generate more than one subgoal. They are responsible for the tree representation of the general proof, rather than just a linear one. As far as the development of the general proof goes, it should be the case that the user may traverse the tree and extend the general proof at various nodes, or else be able to specify a proof using branching rules. In particular, the user should be

able to specify whether repeated use of a branching rule would be upon all the subnodes, or just a subset of these. Note that if the general proof corresponds to a ω-proof subtree (as shown above), then there is already in principle subsequent branching in the general proof.

What might be an example of a branching rule? An obvious example is that of conjunction, namely when the goal is to prove $\vdash \forall x\ (A(x) \wedge B(x))$ by the constructive ω-rule. This would correspond to the $r+1$th subnode of the recursive prooftree in $PA_{c\omega}$ being $\vdash A(r) \wedge B(r)$, upon which the *rand* rule could be used, breaking down the proof into that of $\vdash A(r)$ and $\vdash B(r)$. Is it always possible to apply such a rule before applying the constructive ω-rule, that is, before using the general proof notation? The answer is that it is not always possible to break down the logical quantifiers before using the constructive ω-rule, rather than within the general proof, for with the case of $\forall x(B(x) \vee C(x))$, there could be a case split in the general proof, with $B(r)$ for some r, and $C(r)$ for others (cf. step linearity in Subsection 8.3.2). An example when this happens is in the proof of $\forall x(even(x) \vee odd(x))$ given in example 4. of Subsection 9.1. Note the special status of $\vee$ in the constructive case (cf. $A \vee \neg A$) (Kreisel, 1965, P166):

> "this wholesale carry over of derived rules from predicate logic is one of the special virtues of cut free infinite proofs. An important exception is this: if $A \vee B$ is provable in predicate logic either A or B is provable. In arithmetic, only if A and B are *closed*."

6.6.2 Conditional Rules

There is a further question about how conditional rewrite rules may be applied in a general proof. Note that it is important not to confuse conditional rewrite rules of the form condition $\rightarrow$ rewrite, eg. $x \neq 0 \rightarrow p(s(x)) = x\ *$, with rules which just alter the syntax of the current general proof goal, eg. $true \wedge A \Rightarrow true$. The latter are proof transformations on the general proof which take the form $A \wedge B \Rightarrow A, B$, $A \vee B \Rightarrow A$ (or B) — these account for the case split, $\neg\neg A \Rightarrow A$, $A \vee \neg A \Rightarrow true$, etc. These rules include the other normalisation analogues and might be used to

put the general proof into normal form, as a method of quickly detecting truth or falsity, and are proof transformations rather than derived inference rules.

Given a general proof Q, the new line of the general proof using the conditional rule $\mathsf{cond} \rightarrow x = y$ will be $(\neg \mathsf{cond} \wedge Q) \vee (\ \mathsf{cond}\ \wedge Q[y/x])$. In the case of the 'conditional' example given in Subsection 8.1.5 (namely proof of $\forall x(x \neq 0 \rightarrow p(x) + s(s(x))) = s(x) + x$), when using the rule $*$ given above (ie. 8.12), the new line of the general proof would be: $s^{n-1}(0) = 0 \wedge ((s^n(0) = 0 \vee \underbrace{p(s^n(0)) + s \ldots}_{continues}) \vee (s^{n-1}(0) \neq 0 \wedge \ldots))$. Using the $\vee$-rule twice, equality would be obtained. Thus, the form of the derived rule used when the general proof matches the rewrite part of the conditional rule will be:[3]

$$\frac{\Gamma \vdash \forall \vec{x}(cond(\vec{x}) \rightarrow R(\vec{x}) = S(\vec{x})) \qquad \Delta \vdash T(R(\vec{p}))}{\Gamma, \Delta \vdash (\neg cond(\vec{p}) \wedge T(R(\vec{p}))) \vee (cond(\vec{p}) \wedge T(S(\vec{p})))}$$

6.7 Conclusions

In this chapter the relation of the implementational approach to the theoretical notions of effective trees given in Chapter 5 has been considered; such trees selected by the implementation are shown to be correct, according to a given way of checking for correctness, which involves using induction over various datatypes. To reason about such trees, I work in a theory of trees and of the original syntax (which may be defined effectively). Defining equations for primitive recursive functions are taken as axioms, with a derived form as inference rules to allow rewriting of a formula in the obvious way. Furthermore, the system encapsulated by the implementation (further described in Chapter 10) has been shown to be sound with respect to the system $PA_{c\omega}$ introduced in the previous chapter, although completeness has been left as an open question. Finally, casesplits and conditionals in the general proof have been shown to correspond to branching in the effective tree representation.

---

[3]Note that this would only be an appropriate rule to be used in a classical system, for there is an implicit assumption of $A \vee \neg A$.

Chapter 7

Theoretical Considerations

> *"The method of mathematical induction may be thus described. We prove that if a theorem is true in one case, whatever that case may be, it is true in another case which we may call the* next *case; we prove by trial that the theorem* is *true in a certain case; hence it is true in the next case, and hence in the next to that, and so on; hence it must be true in every case after that with which we began."*
>
> *Todhunter*

The following sections detail:-

- consideration of certain propositions not provable in $PA \setminus cut$;
- clarification of the status of induction as a derived rule within $PA_{c\omega}$;
- the use of meta-induction over proofs;
- an investigation of the relationship between various methods of proof of arithmetic propositions.

7.1 Propositions Not Provable in $PA \setminus cut$

In this section attention will be drawn to certain propositions that seem not be provable in $PA \setminus cut$, and a suggested example of this type will be presented. Such propositions must undergo generalisation in order for a proof to be provided within PA. It will be shown later that for some of these propositions, a proof in $PA_{c\omega}$ may suggest an appropriate cut formula upon which induction may be performed, and hence a proof in PA.

Ever since Gentzen's Hauptsatz, which proved that the cut rule could be eliminated from predicate calculus (Gentzen, 1969), attention has been focussed on whether cut elimination holds for other logical systems; some of the results have already been discussed in Subsection 2.3.5. It is a known result that cut elimination cannot be proved for PA, and that it is the induction rule which poses the problem, rather than any other addition used to form this extension of predicate calculus (Schwichtenberg, 1977). If one restricts PA to a subsystem in which the induction rule is restricted to Σ_1-formulae[1], then cut reduction may be carried out, but only down as far as Σ_1-formulae, because then the rule of induction makes the reduction process come unstuck at the point:

$$\dfrac{\Gamma, A(y) \vdash C \qquad \dfrac{\Gamma \vdash A(0) \quad \Gamma, A(x) \vdash A(s(x))}{\Gamma \vdash A(y)}\, ind}{\Gamma \vdash C}\, cut$$

There is however a partial cut elimination theorem for PA which is provable by adapting the cut-elimination proof for predicate calculus, namely that any provable sequent will be provable without free cuts[2] (Takeuti, 1987, P112). The investigation in this section of certain expressions within PA which may need a cut for their

---

[1] $A(x) \equiv \exists z_1, \ldots, \exists z_n B(x, z_1, \ldots, z_n)$ where B is quantifier-free or, at worst, contains only "bounded" universal quantifiers.

[2] A cut is free if its cut formula is free; a formula A is free if either it has no ancestor which is an induction formula, or it has an induction formula as an ancestor but a logical symbol is introduced in an ancestor of A between any such induction formula and A itself.

proof is consistent with Kreisel's demonstration that there is no primitive recursive algorithm for eliminating cuts within PA, so there is no primitive recursive way of transforming a proof containing cuts to a cut-free proof. More specifically, cut elimination in PA is not possible in the sense that there is no recursively enumerable sequence of derived rules such that:

(i) it can be proved in arithmetic that the rules satisfy the subformula property, and

(ii) the proof figures are finite, so that induction can be applied.

The solution as far as cut elimination is concerned is to weaken (ii) by using infinite proof figures, and use the ω-rule, as has been discussed previously. A different form of induction may be used (see Section 7.2). As mentioned before, there will be cut-elimination in this new system.

However, considering the system $PA \setminus cut$, is it possible to show that certain expressions, such as $\forall x.(x + x) + x = x + (x + x)$, are not provable within this system? One approach is to reduce the problem using the subformula property.[3] The latter states that if $\Gamma \vdash \Delta$ is derivable in predicate calculus, then there is a derivation of this sequent which contains only subformulae of $\Gamma \vdash \Delta$. Hence, if $\Gamma \vdash \Delta$ does not contain $\neg$ or $\rightarrow$, then each formulae in an antecedent of a sequent in the derivation is a subformula of a formula in Γ, and similarly succedent with respect to Δ (since each of the rules has the subformula property). This procedure will eliminate all the logical connectives, leaving only equations (with the rule of induction, plus various axioms), so long as the original sequent did not involve $\neg$ or $\rightarrow$. If there is a cut in the proof, obviously the subformula property no longer holds.

The rules which remain are induction, plus the equality axioms $\vdash s = s$, $A = B \vdash s(A) = s(B)$ and $A = B, A = C \vdash B = C$. Substitution may be considered to be a derived rule, which requires cut, and so is not allowed, but it is more natural to allow it as a basic rule. The use of recursive equations as rewrites must also be part of the system (and these rely on substitution), so in order not to be too severe,

---

[3]This has been discussed in Subsection 2.3.2, with reference to Table 2–1.

substitution will be allowed as a basic rule. The proof of $\forall x\ (x+x)+x = x+(x+x)$ will be blocked in $PA \setminus cut$, since after normalisation when all applicable rules have been used, the only remaining option is to show analyticity of the sequent, which is not possible. (Analytic propositions are those which are equivalent, or definable in terms of each other.) The proof can only be of the form:

$$\frac{\vdash (t+t)+t = t+(t+t)}{\forall x\ (x+x)+x = x+(x+x)} \forall -r$$

or:

$$\dfrac{\dfrac{\ldots \vdash s((r+s(r))+\overbrace{s(r)}^{\theta}) = s(r+s(r+s(r)))}{\vdash (0+0)+0 = 0+(0+0)\ \ (r+r)+r = r+(r+r) \vdash (s(r)+s(r))+s(r) = s(r)+(s(r)+s(r))}\ ind}{\forall x\ (x+x)+x = x+(x+x)}\ normalisation$$

The latter uses rewrite rules $s(x)+y \Rightarrow s(x+y)$ and $0+y \Rightarrow y$ to achieve normalisation of $\vdash (r+s(r))+s(r) = r+s(r+s(r))$, which is still not an equality. These proofs seem to be blocked, but in fact it is possible to carry out some less obvious inductions and substitutions, such as rewrite rules in non-wave directions, and so this discussion forms only an indication that such a proposition might not be provable in $PA \setminus cut$, rather than a concrete proof. However, it might be possible to actually obtain a proof for this particular example by means of the use of a theorem-prover to perform an exhaustive search to see if the process did terminate, for detection of subsumed applications could be used to try to counter the fact that the induction rule may be continually applied.

7.2 Induction Within $PA_{c\omega}$

In this section the nature of induction within $PA_{c\omega}$ will be investigated. In addition, meta-induction over proofs will also be considered. It is important within the following subsections to retain this distinction between induction as a derived rule of inference and the rule of proof which shall be called meta-induction. Section 8.3 provides further discussion regarding the relationship between induction in PA, induction in $PA_{c\omega}$ and general proofs.

7.2.1 Induction as a Derived Rule

Induction is a derived rule in the sense that any use of induction can be captured by use of the ω-rule. The principle of induction is a consequence of the ω-rule, by the following argument: given $\vdash A(0)$ and $\vdash \forall x\ (A(x) \rightarrow A(s(x)))$, then by n-fold application of $\forall-$ and $\rightarrow -$ elimination, we may deduce, for each n, a proof of $\vdash A(\underline{n})$. Hence, by the ω-rule, $\vdash \forall x A(x)$. However, this justification makes essential use of induction (via the cut-rule) for the step concluding $\vdash A(s(\underline{n}))$ from $\vdash A(\underline{n})$ and $A(\underline{n}) \vdash A(s(\underline{n}))$, which possibly shows that the ω-rule is not central to our conception of the structure of the natural numbers (Isaacson, 1992, P21). For in order to prove each $A(\underline{k})$, cut is used to obtain access to $\forall x(A(x) \rightarrow A(s(x)))$ in each branch. See Figure 7–1 for a graphical representation of this argument. Although this is an inessential use of cut in the sense that the proof could be formulated differently (ie. without a cut, by the cut elimination theorem), this demonstrates that even without the explicit use of induction, the latter is being used implicitly because the proof is being specified in a primitive recursive way. The transformation of Figure 7–1 into a proof not involving the cut rule is carried out in an analogous manner to the cut elimination proof for $PA_{c\omega}$ (see Subsection 2.3.5); the cases will differ according to the connectives in A, but cut elimination may be provided for the part of the proof from $A(s(\underline{k}))$ to the open leaf $A(\underline{k})$; this will prove of relevance when considering linearisation of general proofs for the purposes of generalisation in Section 8.3.

$$\dfrac{\Gamma \vdash A(0) \ldots \qquad \dfrac{\vdash \forall x(A(x) \to A(s(x))) \quad \dfrac{\dfrac{}{\Gamma,\ A(s(\underline{k})) \vdash A(s(\underline{k}))}\ axiom \qquad \begin{array}{c}\Gamma \vdash A(0)\\ \vdots\ similarly\\ \Gamma \vdash A(\underline{k})\end{array}}{\dfrac{\Gamma,\ A(\underline{k}) \to A(s(\underline{k})) \vdash A(s(\underline{k}))}{\Gamma,\ \forall x(A(x) \to A(s(x))) \vdash A(s(\underline{k}))}\ \forall l}\ \to l}{\Gamma \vdash A(s(\underline{k}))}\ cut \qquad \ldots}{\Gamma \vdash \forall x A(x)}\ \omega\ rule$$

Figure 7–1: Derivation of Induction in $PA_{c\omega}$: *Given the ω-rule and $\Gamma \vdash A(0)$ and $\vdash \forall x(A(x) \to A(s(x)))$, it is possible to prove with the use of the cut rule that $\Gamma \vdash \forall x A(x)$*

Note that induction in $PA_{c\omega}$ involves the provability of $P(n)$ rather than the proof of $P(n)$, that is to say that $prov(\Pi_1, P(0)), prov(\Pi_2, P(n) \to P(n+1)) \Rightarrow prov(K(\Pi_1, \Pi_2), \forall x P(x))$, where K combines the prooftrees Π_1 and Π_2 in such a manner as may be derived from the associated graphical representation of the effective prooftree given in Figure 7–1 (that is, by instantiating variables in free occurrences of Π_2, and conjoining these prooftrees). The claim is that there is some way of filling in the proof from $P(n)$ to $P(n+1)$ (ie. via K, the combination of the individual proofs Π_1 and Π_2), and that this is the same in all cases. This corresponds to the fact that the proof of $\Gamma \vdash A(s(n))$ may be reduced to that of $\Gamma \vdash A(n)$. If so, this would result in the overall structure of Figure 7–1 as being:

$$\dfrac{\begin{array}{ccccc}\Gamma \vdash A(0) & & & & \\ & & \Gamma \vdash A(0) & & \Gamma \vdash A(\underline{1}) \\ & & \vdots & & \vdots \\ \Gamma \vdash A(0) & \Gamma \vdash A(\underline{1}) & & \Gamma \vdash A(\underline{2}) & \textit{etc.}\end{array}}{\Gamma \vdash \forall x\ A(x)}\ \omega - rule$$

Further discussion of the relationship between induction in PA and the 'linear' form of such a prooftree is given in Section 8.3.

In conclusion, the rule of induction in $PA_{c\omega}$ states that if $PA_{c\omega}$ proves $\Gamma \vdash A(0)$ and also proves $\vdash \forall x(A(x) \to A(s(x)))$, then $PA_{c\omega}$ proves $\Gamma \vdash \forall x A(x)$, which can be shown to be a derived rule.

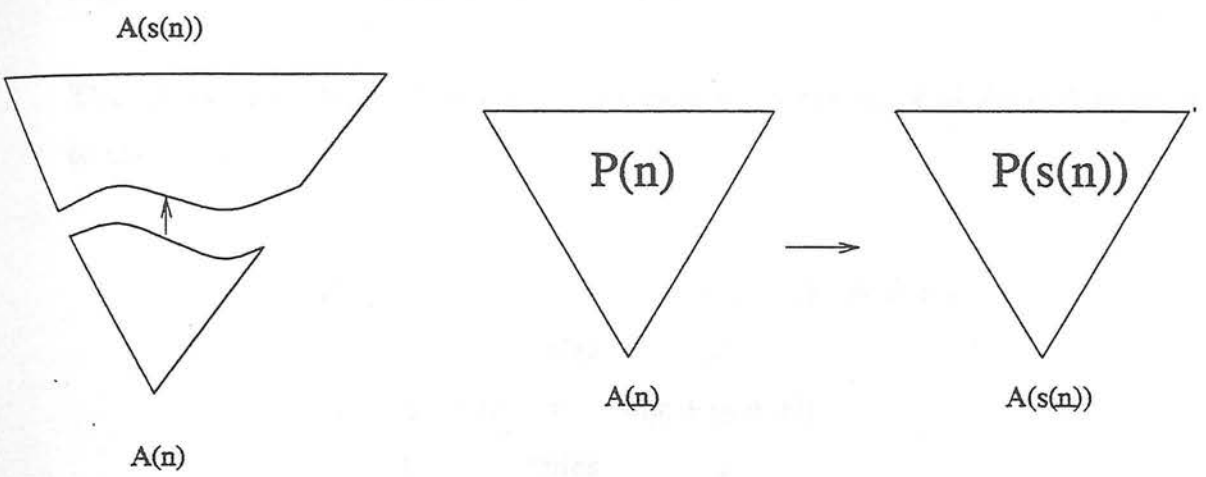


Figure 7–2: Comparison of Standard Induction and Induction over Proofs

7.2.2 Meta-induction Over Proofs

This section discusses meta-induction over proofs. Induction over verification proofs has already been introduced in Sections 5.4 and 6.4.1. This form of meta-induction carries out induction over the natural numbers in the theory of prooftrees and syntax which has been described in Chapter 5. Note that this form of induction is not the same as the derived rule of induction discussed in the previous subsection. Using the Gödel encoding method presented in Subsection 4.3.2 however, these inductions merge, in an analogous manner to primitive recursion and effectiveness (cf. Chapter 5). Meta-induction within $PA_{c\omega}$ is a rule which makes an assertion about proofs, and not about sequents. Ordinary induction over the natural numbers is of the form ($\Gamma \vdash A(0)$ and) $\Gamma, A(n) \vdash A(s(n))$ for arbitrary $n \in \omega$, justifies $\Gamma \vdash \forall x\ A(x)$. In other words, given a hypothesis $A(n)$, the aim is to show the conclusion $A(s(n))$. This rule can be mimicked at the meta-level in $PA_{c\omega}$. The idea, as shown in Figure 7–2, is that meta-induction enables proof transformations of the form "proof of $A(\underline{n})$ justifies proof of $A(s(\underline{n}))$" to result in the conclusion that $\forall x A(x)$ holds. (The n achieves the status of a meta-variable, in this case ranging over numerals, as mentioned earlier.) Hence this induction rule

captures the idea that a proof for $\forall x A(x)$ may be obtained if a proof of $A(s(x))$ is given in terms of extending the proof of $A(x)$.

The following example illustrates such a case when the proof of $A(s(x))$ reduces to the proof of $A(x)$:

$$\begin{array}{ccccc} (s(x)+y)+z & = & s(x)+(y+z) & \mathbf{A(s(x))} \\ \downarrow & \textit{rules} & \downarrow & \\ s((x+y)+z) & = & s(x+(y+z)) & \\ \downarrow & \textit{rules} & \downarrow & \\ (x+y)+z & = & x+(y+z) & \mathbf{A(x)} \end{array}$$

In actual fact, this idea of extension needs to be revised such that $PA_{c\omega}$ proves $A \vdash B$ means that "given a proof of A, one may construct a proof of B". The idea of extension is no longer directly incorporated into the proof, but there are cases in which one would wish to construct a proof of B by using the proof of A as a guide, rather than by a direct extension of A. (In fact, if the definition were not revised in this way, this meta-induction rule would essentially become redundant.) Moreover, as shall be shown below, this form of induction may work when ordinary induction would require use of the cut rule in the proof in order to carry through.

Meta-induction over proofs is a more powerful form of induction than induction in PA, in the sense that there are cases in which induction in PA is blocked (and therefore a cut is required), but meta-induction in $PA_{c\omega}$ works, making use of proof structure. Therefore, a cut in PA may be avoided by carrying out meta-induction over proofs. Proofs are manipulated and not just the hypothesis, as is the case with standard induction. Uniform manipulation is carried out on the nth case proof to give the $n+1$th case proof; also, the structure of the formal numerals themselves may be exploited to give a proof. This will be considered in more detail in the following subsection.

The Use of Meta-induction in $PA_{c\omega}$

When implementing the system $PA_{c\omega}$, one might wish either to generate proofs in $PA_{c\omega}$ (in cases where this is easier than automation in other systems) or to use these proofs as a guide to enable proofs in other systems.[4] In this subsection, a class of proofs for which there is a problem of automatic derivation in PA are considered,[5] but which may be proved in $PA_{c\omega}$. Consider the example $\forall x\ (x + x) + x = x + (x + x)$, for which a general proof has been provided in Figure 6–1. Induction in PA is blocked, because $P(s(x))$ may not be given in terms of $P(x)$, as shown in Section 7.1. However, meta-induction in the system $PA_{c\omega}$ does carry through, although induction in PA did not, for the following crucial reason. The proof of the successor case is not an extension of the proof of the case of its predecessor, as the latter may not be substituted into the former, but instead we may use the rules for the latter as a "guide" for the rules of the former. So the proof of the successor case is described by a uniform transformation on the proof of the previous case, but this transformation is not restricted to extension of the previous proof. This extension of the previous proof is the ω-analogue of induction.[6]

To illustrate — we wish to prove:

$$\underbrace{(\underline{n}+\underline{n})+\underline{n}=\underline{n}+(\underline{n}+\underline{n})}_{\mathcal{A}} \vdash_{PA_{c\omega}} \underbrace{(s(\underline{n})+s(\underline{n}))+\overbrace{s(\underline{n})}^{\mathcal{C}}=s(\underline{n})+(s(\underline{n})+s(\underline{n}))}_{\mathcal{B}}$$

$\mathcal{A}$ may not be substituted directly in a proof of $\mathcal{B}$, because terms like $\mathcal{C}$ (essentially unaltered by the rules given) must be distinct from the other terms in the formula,

---

[4] See Chapter 8.

[5] The problems involved are discussed in Section 7.1. (Notably, a cut elimination procedure cannot be defined for PA.)

[6] The advantages of this method will be of interest later in the thesis, since manipulation of general proofs via meta-induction is closely related to the linearisation process described in Section 8.3, in which the same rule blocks are repeated to reduce $A(s(r))$ to $A(r)$ in the general proof, down to $A(0)$. As shall be shown, this process justifies $\forall x A(x)$, suggesting that the latter would be a suitable cut formula.

in the sense that they must have a distinct variable. This is because one would not wish to carry out induction on these "unaltered terms" as one would wish them to remain the same. In this case this is not so, and hence standard induction does not work. (Note the correspondence between $\mathcal{C}$ above and θ from Section 7.1.)

However, rules for proving $\mathcal{A}$ may be converted to those for proving $\mathcal{B}$ by the "meta-rule" $rule_i(n) \Rightarrow rule_i(s(n))$. Any uniform manipulation on the nth case proof (ie. proof of $\mathcal{A}$), resulting in the $n+1$th case proof (ie. proof of $\mathcal{B}$) will achieve the desired effect. As a result $\mathcal{B}$ may be proved without substituting $\mathcal{A}$ directly.[7]

More importantly, one is dealing with formal numerals and not numbers in this case, and so the structure of the formal numerals in $\mathcal{B}$ may be exploited, to allow use of rewrite rules not previously applicable. This consideration also enables the proof of the original goal by means of the constructive ω-rule.

In conclusion, in cases of this type, induction at the meta-level may be used when induction in PA fails. (For those examples where induction in PA carries through, then a reduction of $P(s(x))$ to $P(x)$ will be possible, and by this same process, induction at the meta-level will carry through as well). Thus, the existence of the logic $PA_{c\omega}$ is justified in theorem-proving terms, for by this means a proof may be given of those propositions not obviously provable in PA without using the cut rule (the latter being considered undesirable as regards the automation of proofs), and for which induction is blocked. In Chapter 8 I move on to consider how this proof might suggest the form of the proof in PA (using cut).

---

[7]Either the rules, given by the nth case proof, may be applied to the consequent, with $n+1$ substituted for n, and the whole method checked as outlined above, or alternatively, one could start with the axioms and use induction to generate rewrite rules of an appropriate form, and then use these rules. For example, given axioms 6.2 and 6.1, as shown in Subsection 6.1.1 induction may be used on the first argument to obtain

$$\forall n\ s^n(0) + y = s^n(0 + y)$$

In fact, in this case, this is the only rule which would be generated, by induction on x. Induction on y would not generate useful new rules. The rule could be changed to apply to the $n+1$th case proof to $\forall n\ s^{n+1}(0) + y = s^{n+1}(0 + y)$, which is exactly the rule which is needed.

7.3 Conclusions

An informal argument has been given that certain propositions not provable in PA without the use of the cut rule (even allowing substitution as a basic rule) are provable within $PA_{c\omega}$. In addition, induction as a derived rule and the use of meta-induction within $PA_{c\omega}$ have been considered.

Chapter 8

A New Method of Generalisation within Automated Deduction

"Generalisation is the process through which we obtain what are called general or universal notions."

Sir W. Hamilton

The next two chapters deal with an application of the implementation of $PA_{c\omega}$ in the field of generalisation, which, as discussed in Chapter 3, is very important from a theorem-proving point of view. This chapter presents a new method of generalisation by means of which proofs in $PA_{c\omega}$ may automatically guide proofs in Peano Arithmetic. (The implementation of this algorithm is discussed in Chapter 10.) The possibility of extending this generalisation method to other domains is shown in Section 8.2. The methods of using explanation-based generalisation and linearisation on the individual subtrees of the proofs of the ω-rule, such that a cut formula is suggested upon which induction may be performed, are expounded in Sections 8.1.4 and 8.4. Methods of generalisation already proposed by other people work on most of the examples presented in this chapter, but there are some examples, particularly in the following chapter, for which the proposed generalisation method suggests a generalisation when others do not.

As discussed in Section 3.1, there are many different types of generalisation, ranging from the generalisation of the proposition A to $A \wedge B$ and, for the predicate P, $\exists x \exists y\ P(x,y)$ to $\exists x\ P(x,x)$, or $\forall x \forall y\ P(x, f(y))$ to $\forall x \forall a\ P(x,a)$. In this chapter

the following generalisation will be concentrated upon: $\forall x\ P(x,x)$ to $\forall x \forall a\ P(x,a)$; in the following chapter $\forall x\ P(x,c)$ to $\forall x \forall a\ P(x,a)$ will also be considered. Current methods of generalisation are presented in Section 11.3. My approach will not be specific to sequent calculus since, as discussed in Subsection 3.1.1, there are analogous problems in other presentations.

8.1 How Proofs in $PA_{c\omega}$ Suggest Generalisation in PA

8.1.1 General Approach

There is a class of proofs, including $\forall x\ (x+x)+x = x+(x+x)$, which appear to be provable in PA only using the cut rule but which are provable in $PA_{c\omega}$ (see Section 7.1). In this section I consider whether the proof in $PA_{c\omega}$ suggests a proof in PA, and so in particular, what the *cut formula* would be in a proof in Peano Arithmetic. That is, for the example mentioned above, I would wish to suggest a suitable A which will enable the completion of the proof, such that:

$$\frac{A \vdash \forall x\ (x+x)+x = x+(x+x) \qquad \vdash A}{\vdash \forall x\ (x+x)+x = x+(x+x)^*}\ CUT$$

Ordinary induction does not work on $*$; this is discussed in more detail in Section 7.1). That is why it is necessary to use the cut rule. One would wish A to be a more general version of $*$, so that $A \vdash *$ could be proven, but on the other hand to be suitable to give an inductive proof, so that $\vdash A$ could be proven by induction. Hence the problem of generalisation would be tackled by using an alternative (stronger) representation of arithmetic as a guide.

A heuristic for generalisation would be to examine what remains unaltered in the nth case proof (or "general proof") and then write out the original formula, but with the corresponding term re-named. What is meant by 'unaltered' is defined by what is unaffected by the rewrite rules. The terms should be renamed because one would not wish to carry out induction upon these terms, since the general proof

shows that they are not altered within the proof process, unlike an induction variable. The heuristic may be regarded as a simplification of the explanation-based generalisation method considered in Subsection 8.1.4. This method works both for the cases of generalising terms to variables and variables apart, as shown in Subsection 8.1.3, but, as shall be shown in Section 8.4, it requires modification in the case of the addition of accumulators.

8.1.2 Strategy for Normalisation

It is important at this stage to clarify the strategy for generation of a general proof, since the rewrite rules applied will affect the generalisation suggested. The algorithm used in the implementation for this stage is discussed in Section 10.4. If the "unaltered term" heuristic is to be used, it is important not to apply just any rewrite rule which is applicable to the general proof, but rather those sufficient to reach equality in the general proof. In other words, once equality is reached in the general proof, additional rules may still be applied, but by so doing it may be the case that a more general proof will not be provable with the same rule set, and hence that the explanation-based generalisation method will be too constrained to suggest a new initial formula. Hence, for the purposes of generalisation, the general proof of interest is the minimal general proof which results in equality (in other words, such that redundant rules not needed in order to give equality are not included).[1] In practice, all that is required is an equality check on each line of the general proof, which terminates further rewriting if it succeeds: this check is present in the implementation. The effect of rearrangement of rewrite rules within the general proof is considered in Section 8.4: if rearrangement without addition of rewrite rules provides a proof of an identical inital sequent, then the same generalisation would be suggested — otherwise, this would not necessarily be the case.

---

[1]Note the connection between such a restriction of the search space and the way in which other generalisation methods choose between possible solutions (such as Jane Hesketh's exploration of multiple rewrite rule applications (Hesketh, 1991b)).

8.1.3 Examples Illustrating Application of the Suggested Heuristic

A method of generalisation has been proposed. This section investigates how it fares in practice. This method covers the cases when generalisation from terms to variables and also generalisation of variables apart is normally used; an extension of the method to deal with generalisation of constants to variables and introduction of accumulators shall be discussed in Section 8.4.

Generalisation of Variables Apart: $\forall x\ (x+x)+x = x+(x+x)$

For the example corresponding to the general proof of $\forall x\ (x+x)+x = x+(x+x)$, which is discussed in Subsection 6.1.1 and presented in Figure 6–1, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ remain essentially unaltered. As well as this being apparent from the general proof, this could be deduced using the axioms and the argument positions of $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ in the formula. These terms must therefore not be involved in an induction. So, one might rewrite the variable corresponding to $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ as y. In this case, this would give

$$A \equiv (x+y)+y = x+(y+y).$$

A could then be proved by induction on x. Ordinary induction on $\forall x(x+x)+x = x+(x+x)$ is blocked, and yet induction on the suggested generalisation carries through:

$$\begin{array}{rcl} (x+x)+x = x+(x+x) & \vdash & (s(x)+s(x))+s(x) = s(x)+(s(x)+s(x)) \\ & \ldots \vdash & s((x+s(x))+s(x)) = s(x+s(x+s(x))) \end{array}$$

blocked

$$\begin{array}{rcl} (x+y)+y = x+(y+y) & \vdash & (s(x)+y)+y = s(x)+(y+y) \\ & \ldots \vdash & s((x+y)+y) = s(x+(y+y)) \end{array}$$

equality, by weak fertilisation

Generalisation of Terms to Variables: $x+s(x) = s(x)+x$

This is Example 8.3 from Table 8–1, and is another example of a proposition which requires cut in its proof in PA. Let us consider whether the formula is

provable in $PA_{c\omega}$; there is a general proof of the nth case, thus allowing proof by the constructive ω-rule.

$$\begin{array}{rcll} s^n(0)+s(s^n(0)) & = & s(s^n(0))+s^n(0) & \\ s^n(s(s^n(0))) & = & s(s^n(s^n(0))) & \textbf{see Note 1} \\ s^n(0) & = & s^n(0) & \textbf{see Note 2} \\ & EQUALITY & & \end{array}$$

Note 1: from Axioms 6.2 and 6.1, one may derive $s^n(0)+z = s^n(0+z) = s^n(z)$ (cf. proof in Chapter 6 of $s^n(x)+y = s^n(x+y)$).
Note 2: it is necessary to know $s(s^n(X)) = s^n(s(X))$, which may be shown by cancellation. Note that $s^r(X)$ is shorthand for $\underbrace{s \ldots s}_{r \text{ times}}(X)$.

In order to address the question of how a proof using ordinary induction might be suggested, let us consider the generalisation heuristic. This requires careful consideration of which terms remain unchanged by the rewrite rules.

$$\begin{array}{rcll} x+s(x) & = & s(x)+x & x \rightarrow s^n(0) \\ \underbrace{s^n(0)}_{\mathcal{A}}+\underbrace{s(s^n(0))}_{\mathcal{B}} & = & \underbrace{s(s^n(0))}_{\mathcal{C}}+\underbrace{s^n(0)}_{\mathcal{D}} & \mathcal{B},\mathcal{D} \textit{ retain position and form} \\ s^n(0+\underbrace{s(s^n(0))}_{\mathcal{B}}) & = & s(s^n(0+\underbrace{s^n(0)}_{\mathcal{D}})) & \textit{Ditto} \\ s^n(\underbrace{s(s^n(0))}_{\mathcal{B}}) & = & s(s^n(\underbrace{s^n(0)}_{\mathcal{D}})) & \textit{Equality by cancellation} \end{array}$$

Hence $\overbrace{x}^{changed} + \underbrace{s(x)}_{unchanged} = \overbrace{s(x)}^{changed} + \underbrace{x}_{unchanged}$

The generalisation suggested is therefore $x+s(y) = s(x)+y$, since it is not desirable to do induction on $\mathcal{B}$ or $\mathcal{D}$. The generalisation carries through, even though it is not one which might obviously spring to mind. This is also the generalisation suggested by the (more rigorous) explanation-based generalisation method. This cut formula enables a proof by induction in PA, as follows:

$$\dfrac{\dfrac{\dfrac{}{\forall x \forall y\ x + s(y) = s(x) + y \vdash \ldots}\ Subst, y = x \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{r + s(y) = s(r) + y \vdash r + s(y) = s(r) + y}\ EQUALITY}{r + s(y) = s(r) + y \vdash s(r + s(y)) = s(s(r) + y)}\ \pi}{r + s(y) = s(r) + y \vdash s(r) + s(y) = s(s(r)) + y}\ 6.2}{\vdash \forall x \forall y\ x + s(y) = s(x) + y}\ base\ case\ ind(x)}{\vdash \forall x\ x + s(x) = s(x) + x}\ cut$$

$\pi : A = B \Rightarrow s(A) = s(B)$

In conclusion, generalisation of variables apart is carried out by the method suggested in order to deal with cases in which the generalisation of common subexpressions would be used by some other methods; for instance, for this example, many methods would just initially generalise to $\forall x \forall y\ x+y = y+x$ (cf. Chapter 11). However, in some cases the effect is the same in the end (cf. Subsection 8.4.4).

It has been seen that the suggested heuristic works for simple arithmetical examples, at least. A cut formula is suggested, but it is not necessarily the most general one. Now, the rules used in the general proof are the same for both $\forall x\ (x + x) + x = x + (x + x)$ and $\forall x \forall y \forall z\ (x + y) + z = x + (y + z)$. This is because the latter is a generalised form of the former and this fact may be exploited when doing such proofs. There exists a more general method which produces the most general cut formula obtainable from using the same rules as in the general proof. This is explanation-based generalisation (see Sections 3.4 and 8.1.4, and for example (Donat & Wallen, 1988)), applied to this new domain. This may only produce a correct cut formula if the structure of the cut formula is the same as that of the original goal; if additional structure such as accumulators are required, modification of the method is required, as shall be seen in Subsection 8.2.2. This procedure is amenable to automation, and has been implemented, so that a cut formula may be produced automatically from a general proof (see Chapter 10). Examples of a cut formula which may be suggested by this method of generalisation but not alternative ones are given in Chapter 11. The explanation-based generalisation method of proof for this example is given in detail in the following section.

8.1.4 Explanation-Based Generalisation

In this section the explanation-based method used is described in more detail. Section 3.4 provides a background to this approach, mentioning other systems using various domains including integration problems. In general terms the process works by generalising a particular solution to the most general possible solution which uses the rules of the original solution. It does this by applying these rules, making no assumptions about the form of the generalised solution, and using unification to fill in this form.[2]

The method is applied in this instance to a new domain, namely that of general proofs. For the purposes of illustration I shall first consider the example given in Section 8.1, namely the proof of $\forall x\ (x + x) + x = x + (x + x)$. The general proof is repeated below from Figure 6–1, with the difference that the subpositions to which each rule is applied are given here; relevant rewrite rules used are those defining addition, namely $rule1 : s(X) + Y \Rightarrow s(X + Y)$ and $rule2 : 0 + Y \Rightarrow Y$, corresponding to axioms 6.2 and 6.1. The rules applied are given in the notation described in Section 6.1: the first argument of the rule is the subposition of the expression corresponding to the domain of its application, using the notation described in Appendix B.3 (the relevant subposition is underlined in the general proof for clarification) and the second argument is the number of times the rule is applied. Thus, the general proof as written is a (shortened) description of the whole proof, since it just indicates the application of whole blocks of rewrite rules.

$$
\begin{array}{rcll}
(\underline{n} + \underline{n}) + \underline{n} & = & \underline{n} + (\underline{n} + \underline{n}) & \text{Rules} \\
(\underline{s^n(0) + s^n(0)}) + s^n(0) & = & s^n(0) + (s^n(0) + s^n(0)) & rule1([1,1], n) \\
s^n(\underline{0 + s^n(0)}) + s^n(0) & = & s^n(0) + (s^n(0) + s^n(0)) & rule2([2,2,1,1], 1) \\
s^n(s^n(0)) + s^n(0) & = & \underline{s^n(0) + (s^n(0) + s^n(0))} & rule1([2], n) \\
s^n(s^n(0)) + s^n(0) & = & s^n(\underline{0 + (s^n(0) + s^n(0))}) & rule2([2,2,2], 1) \\
\underline{s^n(s^n(0)) + s^n(0)} & = & s^n(s^n(0) + s^n(0)) & rule1([1], n) \\
s^n(s^n(0) + s^n(0)) & = & s^n(s^n(0) + s^n(0)) & equality
\end{array}
$$

As mentioned earlier, a simplified approach towards generalisation would be to look at the proof to see which terms remain unchanged, and generalise these to a

[2]This process has already been illustrated in Figure 3-1.

new variable. A further development is to instead look at the rules of the general proof, and work out what the most general statement could be which was proved using these rules. This is the approach of explanation-based generalisation, and would justify that the cut formula suggested was indeed a proper generalisation.

By the 'unaltered variable' method, a cut formula is suggested of $\forall x\ (x+?)+? = x + (?+?)$, where the ?'s may be rewritten as the same new variable, or different variables using some sort of heuristic, such as that the nth new variable of the left hand side of the equation would be set equal to the nth new variable on the right, depending on whether simply a generalisation, or the most general generalisation, was required. This would suggest cut formulae of

$$\forall x\ (x + y) + y = x + (y + y)$$

or

$$\forall x\ (x + y) + z = x + (y + z)$$

respectively.

Now let us construct the generalisation using the rules R, for this example. The right hand column are the instantiations of variables, which are finally to be filtered back up into the original expression. Essentially what is happening is that the application of the rules are matched to see what the generalisation could be. So if 6.2 is applied m times, this will match with the form

$$s^m(X) + Y \Rightarrow s^m(X + Y)$$

The process is given in Figure 8–1. Nothing more is supposed about the original form of the general proof than that it is of the form $U = W$. The rule application blocks on the left hand side of this figure are identical with those of the general proof given in the preceding paragraph. The procedure is to form the most general proof which could use those same rules to achieve equality. Hence, these same rewrite rules are applied at the specified subpositions to give a new general proof. In so doing the structure of U and W is revealed. For instance, the fact that rule 6.2 may be applied n times at subposition [1,1] of $U = W$ reveals that U must be of the form $fn0([s^n(X)+Y/K])$ (which represents some functor $fn0$ of as

yet unknown arity with initial argument $s^n(X)+Y$ and additional arguments K) before the rule application, and of the form $fn0([s^n(X+Y)/K])$ afterwards. This process is repeated until all the given rules are exhausted. Finally, the left-hand side and the right-hand side of the general proof are unified (since the original proof resulted in equality). Throughout this process, information will have been obtained regarding the structure of some of the postulated variables in this new general proof, such as that presented in the final column of Figure 8–1.

rules	form of general	proof	instantiations
$6.2([1,1],n)$	$fn0([s^n(X)+Y/K])$	$=W$	*original*
$6.1([2,2,1,1],1)$	$fn0([s^n(X+Y)/K])$	$=W$	
$6.2([2],n)$	$fn0([s^n(Y)/K])$	$=W$	$X=0$
$6.1([2,2,2],1)$	$fn0([s^n(Y)/K])$	$=s^n(P+Q)$	$W=s^n(P)+Q$
$6.2([1],n)$	$s^n(Y+K)$	$=s^n(Q)$	$P=0,\ fn0=+$
EQUALITY	$s^n(Y+K)$	$=s^n(Y+K)$	$Q=Y+K$

Figure 8–1: Illustration of Explanation-Based Generalisation on Rules of General Proof

Feeding such variable instantiation information back to the original expression $U=W$ shows that it must be of the form:

$$(\underline{n}+Y)+K=\underline{n}+(Y+K)$$

This gives the most general generalisation as being

$$\forall x\ \forall y\ \forall k\ (x+y)+k=x+(y+k)$$

The whole process is really just a term-matching exercise, and has been successfully automated, as described in Chapter 10. This process has been applied in various other domains, and is the approach of explanation-based generalisation (denoted 'EBG' as an abbreviation).

The EBG method proposed in this section will succeed in the sense that there does exist some general proof such that a correct cut formula could be found by EBG (so long as inductive proof by generalisation apart is possible). However, it will not necessarily work with any given general proof, nor if generalisation apart is not appropriate for the example under consideration.

In conclusion, the heuristic for generalisation suggested is of seeing what remains unaltered in the nth case proof, (viz. the "general proof") and then writing out the original formula, but with that variable re-named. Although this heuristic is suitable for implementation (and was successfully implemented), the method of explanation-based generalisation extends this idea to provide a uniform algorithm based on the underlying structure of the proof. The implementation of EBG described in Appendix E follows the unification process described above, and thus subsumes the implementation of the heuristic method. Note that I do not intend to develop heuristics which, together with this information, might provide a choice of solutions which may be evaluated. I have instead proposed a method which involves less of an *ad hoc* approach, and relies on an analysis of the proofs.

8.1.5 Further Examples which Illustrate How a Cut Formula Might be Provided

The preceding sections have discussed how a method of generalisation may be provided. A selection of further arithmetical examples for which this method works in a straightforward manner is given in Table 8–1 and details of the cut formulae suggested are given in Table 8–2. It may be interesting to consider an example involving multiplication and one involving the use of conditionals plus a destructor function, to illustrate that such extensions do not pose a problem for the method, and because of the questions of cancellation and use of associativity of plus that they highlight.

An Example Involving Multiplication (8.5): $(x + x).x = x.x + x.x$

The axioms to be used are the Peano axioms defining addition and multiplication, namely 6.2 and 6.1 plus:

$$0.y = 0 \tag{8.9}$$

$$s(x).y = x.y + y \tag{8.10}$$

$$\begin{aligned}
\forall x\ x + s(x) &= s(x + x) && (8.1)\\
\forall x\ (x + x) + x &= x + (x + x) && (8.2)\\
\forall x\ x + s(x) &= s(x) + x && (8.3)\\
\forall x\ x.(x + x) &= x.x + x.x && (8.4)\\
\forall x\ (x + x).x &= x.x + x.x && (8.5)\\
\forall x\ (2 + x) + x &= 2 + (x + x) && (8.6)\\
\forall x\ \forall y\ (x + y) + x &= x + (y + x) && (8.7)\\
\forall x\ x \neq 0 \rightarrow p(x) + s(s(x)) &= s(x) + x && (8.8)
\end{aligned}$$

Induction is blocked for the above expressions, but they may all be proved by the method proposed (namely by using the constructive ω-rule) and a correct cut formula produced as appropriate.

Table 8–1: Some Examples of Arithmetical Theorems Proved

Note that axiom 8.10 applied n times corresponds to:

$$s^n(X).Y = ((\ldots(X.Y + \underbrace{Y) + \ldots) + Y)}_{n\ times}$$

(written $X.Y + \underbrace{Y}_{n\ times}$ as shorthand).

Using the axioms 6.2, 6.1, 8.9 and 8.10 given above, the general proof is as follows:

$$\begin{aligned}
(\underline{n} + \underline{n}).\underline{n} &= \underline{n}.\underline{n} + \underline{n}.\underline{n} && \\
(\underline{s^n(0) + s^n(0)}).s^n(0) &= s^n(0).s^n(0) + s^n(0).s^n(0) && 6.2\ n\ times\\
\underline{s^n(0 + s^n(0))}.s^n(0) &= s^n(0).s^n(0) + s^n(0).s^n(0) && 6.1\\
\underline{s^n(s^n(0)).s^n(0)} &= s^n(0).s^n(0) + \underline{s^n(0).s^n(0)} && 8.10\ n\ times\\
s^n(0).s^n(0) + \underbrace{s^n(0)}_{n\ times} &= s^n(0).s^n(0) + (\underline{0.s(0)} + \underbrace{s^n(0)}_{n\ times}) && 8.9,\ 6.1\\
(s^n(0).s^n(0) + \underbrace{s^n(0)}_{n\ times}) &= s^n(0).s^n(0) + (\underbrace{s^n(0)}_{n\ times}) && EQUALITY\ by\ assoc
\end{aligned}$$

The final goal necessitates proof of $(\ldots(s^n(0).s^n(0) + \underbrace{s^n(0)) + \ldots) + s^n(0))}_{n\ occurrences} = s^n(0).s^n(0) + \underbrace{(\ldots((s^n(0) + s^n(0)) + s^n(0))\ldots + s^n(0))}_{n\ occurrences}$. See Section 8.6 for discussion of the use of associativity of plus in such proofs in order to obtain the final equality in the general proof.

The generalisation suggested is

$$\forall x\ \forall y\ (x + y).y = y.y + x.y$$

This is a correct generalisation, as shown below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\textit{CLOSES}}}{\forall y\ (r+y).y = y.y + r.y \vdash \forall y\ y.y + r.y + y = y.y + r.y + y}\ \textit{equality}}{\forall y\ (r+y).y = y.y + r.y \vdash \forall y\ (r+y).y + y = y.y + r.y + y}\ \textit{hyp}}{\forall y\ (r+y).y = y.y + r.y \vdash \forall y\ s(r+y).y = y.y + r.y + y}\ 8.10}{\forall y\ (r+y).y = y.y + r.y \vdash \forall y\ (s(r)+y).y = y.y + s(r).y}\ 6.1}{\vdash \forall x\ \forall y\ (x+y).y = y.y + x.y}\ \textit{ind}$$

Note the difference between this cut formula and the one suggested for the corresponding Example 8.4, which suggests $\forall x \forall y\ x.(y+y) = x.y + x.y$.

By EBG, a more general generalisation of $\forall x \forall y \forall z (x+y).z = y.z + x.z$ is obtained, which is also a correct generalisation.

An Example Involving Conditionals and Destructor Functions (8.8): $\forall x\ x \neq 0 \rightarrow p(x) + s(s(x)) = s(x) + x$

This type of example would pose a problem for most of the other generalisation methods; see Chapter 11 for a comparison. The axioms used will be as follows:

$$x = 0 \ \rightarrow\ p(x) = x \tag{8.11}$$

$$x \neq 0 \ \rightarrow\ p(s(x)) = x \tag{8.12}$$

The general nth case proof will be of the following form (case splits are ignored in order to simplify the presentation):

$$\begin{array}{rcll} \underline{p(s^n(0))} + \underbrace{s(s(s^n(0)))}_{\textit{unchanged}} & = & s(s^n(0)) + \underbrace{s^n(0)}_{\textit{unchanged}} & 8.12,\ 6.1 \\ \underline{s^{n-1}(0) + s.s.s^n(0)} & = & \underline{s(s^n(0) + s^n(0))} & 6.1 \\ \underline{s^{n-1}(0 + s.s.s^n(0))} & = & s.s^n.\underline{(0 + s^n(0))} & 6.2 \\ s^{n-1}(s.s.s^n(0)) & = & s.s^n.s^n(0) & \textit{EQUALITY by cancellation} \end{array}$$

To obtain the general proof, it is only necessary to know that $s^n(0)$ stands for $\underbrace{s \ldots s}_{n\ times}(0)$, and to use cancellation (see Section 8.6 for further discussion on this topic). According to the generalisation method suggested, the unchanged terms are renamed, to avoid induction being applied to them. Hence, the generalisation is:

$$x \neq 0 \rightarrow p(x) + s(s(z)) = s(x) + z$$

and the solution yielded (in PA) is as follows (base case omitted):

$$\dfrac{\dfrac{\dfrac{\dfrac{0 \neq 0 \rightarrow s(s(z)) = s(z) \vdash 0 = s(0)}{\ldots p(0) + s(s(z)) = s(0) + z \vdash 0 = s(p(0))}\,6.1, 6.2 \quad \dfrac{\vdash s(m) = s(m)}{m = s(p(m)) \vdash s(m) = s(p(s(m)))}\,8.12}{r \neq 0 \rightarrow \ldots \vdash r = s(p(r))}\,ind}{\ldots \vdash r + s(s(z)) = s(p(r)) + s(s(z))}\,structural\ rule}{\ldots \vdash r + s(s(z)) = s(p(r) + s(s(z)))}\,6.1}{\ldots \vdash (s(r) \neq 0 \rightarrow) r + s(s(z)) = s(s(r) + z)}\,hyp(subst)}{r \neq 0 \rightarrow p(r) + s(s(z)) = s(r) + z \vdash s(r) \neq 0 \rightarrow p(s(r)) + s(s(z)) = s(s(r)) + z}\,8.12, 8.10}{\vdash x \neq 0 \rightarrow p(x) + s(s(z)) = s(x) + z}\,ind$$

Comment

Note that what is allowing this method to solve many of these examples, which could not be solved by straightforward induction, is that the general proof deals with numerals rather than numeric variables and thus if n was previously in a blocked position with regard to some axiom which could not be applied, the new structure would allow the application of a rewrite rule. As an illustration, for the example $\forall x\ (x+x)+x = x+(x+x)$ considered above, the second, third and sixth occurrences of x are unaffected by the rewrite rule corresponding to axiom 6.2, and hence induction is blocked. The general proof continues from this stage by exploiting the structure of the terms in the first and fourth position, in such a way that further rewrite rules may be applied.

The way in which the general proof often seems to allow extension of the failed (blocked) induction proof, by means of further rewrite rules being applied to additional structure, may be seen for the case of list examples in Appendix C. In Section 8.2 I consider an analogous approach to other domains which would allow solution of generalisation problems.

8.1.6 Conclusions

The generalisation heuristic proposed is to rename terms which are unaltered throughout the general proof, and it works on basic examples by providing an

insight into the actual structure of the proofs. A more general method of performing such a process is provided by explanation-based generalisation. Further discussion of why the formula suggested by EBG is suitable for induction is given in Subsection 8.3.3. Just as the heuristic proposed of renaming unchanged terms is subsumed by the more general method of EBG, the latter in turn shall be shown in Section 8.3 to be an instance of the more general approach of linearisation of general proofs. The following sections develop the algorithms given in this section in order to allow generalisation of more complicated examples in a variety of domains.

8.2 Application of the Proposed Generalisation Method to Domains Other Than Arithmetic

A new approach to generalisation over the natural numbers has been developed above. However, the approach applies more generally to other domains. So long as a general procedure for constructing a proof for each individual of a sort is specified, universal statements about objects of the sort can be proved. The data-types of lists and sets are considered below; the domain of lists is highlighted in particular, in preparation for consideration of examples in the following section of how generalisation may be suggested. An extension of this approach to arbitrary data-types is discussed in Subsection 12.1.3.

8.2.1 The Omega Rule over Various Domains

What form would the ω-rule take for domains other than arithmetic? For the domain of arithmetic, numerals are used in the numerator: a numeral is a term which canonically represents an integer, cf. (Girard, 1987, P47). In the same way, given some canonical representation $\mathcal{L}(n)$ for lists, the ω-rule would take the form

$$\frac{A(\mathcal{L}(0))\ A(\mathcal{L}(1)) \ldots A(\mathcal{L}(n)) \ldots}{\forall l \in \mathit{typelist}\ A(l)}$$

For an arbitrary datatype D, so long as a canonical representation $\mathcal{K}(n)$ may be given of D, then the following ω-rule may be used:

$$\frac{A(\mathcal{K}(0))\ A(\mathcal{K}(1)) \ldots A(\mathcal{K}(n)) \ldots}{\forall x \in D\ A(x)} \omega\ rule$$

8.2.2 Lists

Just as the representation of a natural number n may be seen so be $s^n(0)$, the general representation of a list $\mathcal{L}$ may be seen to be $[x_1, x_2, x_3, \ldots, x_m]$, where m is the length of the list, and the x_i are terms of a specified type eg. the natural numbers. These general representations may be deduced from the definitional equations of the system.

Note that some recursive representation which generates all lists is required. In the same way as the natural number hierarchy $0, s(0), s(s(0)), \ldots$ might be defined as follows:

$$s^0(0) = 0$$
$$s^{s(n)}(0) = s(s^n(0))$$

one may define the list hierarchy $\mathcal{L}(0), \mathcal{L}(s(0)), \mathcal{L}(s(s(0))), \ldots$ where $\mathcal{L}$ is a polymorphic function as follows:

$$\mathcal{L}(0) = nil$$
$$\mathcal{L}(s(n)) = x_{s^n(0)} :: \mathcal{L}(n)$$

Thus there is the structure

$$nil \equiv \mathcal{L}(0)$$
$$x_0 :: nil \equiv \mathcal{L}(s(0))$$
$$x_{s(0)} :: x_0 :: nil \equiv \mathcal{L}(s(s(0)))$$
— and so on.

Of course, $\mathcal{L}(n)$ is not unique, since it only defines a list of length n, and does not stipulate anything about the behaviour of its members. Such information might in some cases influence the structure of the proof, or give different proofs for different

values. As far as equality is concerned, $n = m \Rightarrow s^n(0) = s^m(0)$. However, it is non deterministic whether $n = m \Rightarrow \mathcal{L}(n) = \mathcal{L}(m)$, for these will only be lists of the same length. The definition of this structure may well affect the shape of the general proofs produced; for example, a split-definition into odd or even cases for instance would cause corresponding split cases in the general proof.

In Section 8.5.1 attention is drawn to the fact that for datatypes for which the given structure and function definitions are analogous, general proofs will have an analogous structure. One such relevant factor considered in that section is that the structural definitions should release structure in a similar manner: for example, the numerals could instead be generated by defining equations $\underline{0} = 0$ and $\underline{n+1} = s.\underline{n}$. This corresponds to lazy evaluation, as opposed to the strict evaluation given above, and is analogous to the lazy list definitions given above. The strict definitions tend to be more useful for arithmetic, because arithmetical functions often require the whole structure of a numeral, but list operations nearly always divide the list into head and tail, and hence the lazy definitions provide a more concise expression and may save unnecessary evaluation.

Note that not only is it the case that certain new structural patterns may be seen in the general proof which may guide generalisation, but also that the general representation enables the structure of the particular data-type to be exploited, in the sense that rewrite rules may be used which would not otherwise be applicable. Also, additional structure may be revealed, as shown in the examples given in Section 8.4.

8.2.3 Sets

I shall move on to consider the domain of finite sets. An arbitrary set may be represented by $\mathcal{C}(n)$, where:

$$\mathcal{C}(0) \equiv \emptyset$$
$$\mathcal{C}(s(n)) \equiv \{x_{s^n(0)}\} \cup \mathcal{C}(n)$$

Thus there is the structure (where $\{a\} \cup \{b\} \equiv \{a, b\}$):

$$\begin{aligned}\emptyset &\equiv C(0)\\ \{x_1\} &\equiv C(s(0))\\ \{x_1, x_2\} &\equiv C(s(s(0)))\\ &\vdots\end{aligned}$$

It is important to appreciate that sets are rather like lists, but with the problem of identity statements. For with sets, unlike lists, it is not the case that

$$\{x_i\} \cup S \equiv \{x_j\} \cup S' \Rightarrow x_i = x_j.$$

Since the theory of sets does not have the same uniqueness property as other theories like strings, lists and trees, one has to bear in mind that augmenting the theory with axioms of recursive form may destroy the consistency of the theory. It is necessary to work out some way to deal with equality statements. Rather than doing this via the representation, one may add the identity axioms (for sets S,T and set elements x_i, x_j, where $=$ is the equality between set elements, and $\equiv$ the equality between sets):[3]

$$\begin{gathered}\{x_i\} \cup S \equiv \{x_j\} \cup T \text{ if } x_i = x_j \text{ and } S \equiv T\\ \{x_i\} \cup S \equiv \{x_i\} \cup \{x_i\} \cup S\\ \{x_i\} \cup \{x_j\} \cup S \equiv \{x_j\} \cup \{x_i\} \cup S\end{gathered}$$

Example: $\forall C\ C \cup (\{a\} \cup C) \equiv (\{a\} \cup C) \cup C$

This is very similar to the natural numbers example: $\forall n\ n + (a + n) = (a + n) + n$, plus Example 8.3. The general proof is as follows:

$$\begin{aligned}C(s(n)) \cup (\{a\} \cup C(s(n))) &= (\{a\} \cup C(s(n))) \cup C(s(n))\\ (\{x_{s^n(0)}\} \cup C(n)) \cup (\{a\} \cup (\{x_{s^n(0)}\} \cup C(n))) &= (\{a\} \cup (\{x_{s^n(0)}\} \cup C(n))) \cup (\{x_{s^n(0)}\} \cup C(n)))\\ &\vdots \quad \textit{and so on}\\ \textit{EQUALITY} &\ \textit{by}\ \textit{identity axioms}\end{aligned}$$

---

[3] See, for comparison, Chisholm's representation of sets (Chisholm, 1988).

It is interesting and possible, as shown above, to extend the approach of generalisation suggested via a general proof within the domain of natural numbers to other domains, such as sets, trees and other data types. It can be seen that it might be possible to automatically generate the general representation of an object of a certain type from the system definitions (and then to automatically produce a general proof and use this to guide proofs in the system). The approach described enables the structure of the particular data-type to be exploited (at least for freely constructed types) and a proof to be reached when otherwise it might not have been. Further evidence and examples will be considered later in the chapter.

8.3 Linearisation

In this section the idea of linearity will be discussed, in order to draw a comparison between the linear form of a general proof and the possibility of carrying out induction on the analogous initial formula. This will be seen to provide a more general method of generalisation which may be used when the explanation-based generalisation method is not applicable — for example in the case of accumulator-type generalisation (and which in fact subsumes this method).

8.3.1 The Relationship between Linearity and Induction

In this subsection the relationship between linearity and induction will be considered. First, it is important to clarify the notion of linearity, since it may be used in different senses. Such differences may depend on whether the ω-rule under consideration is constructive: in the case of the constructive ω-rule operating on $\forall x\ A(x)$, the $A(i)$ are uniformly generated, and there will be a relationship between their proofs — otherwise, this may not be the case. The general form of a linear proof involving a single use of the constructive ω-rule is given in Figure 8–2, with the proviso that there may be additional leaves if the $P(i)$ consist

of branching proof rules. Hence proofs are regarded as linear in the sense of their construction, rather than in their representation; that is to say that any ω-proof which is of the form of Figure 8–2 (with additional leaves in the case of inductive proofs with branching in the step proof) is (strongly) *linear*, with the constraints that $A(\underline{k})$ is reduced to $A(\underline{k-1})$ in a uniform way for each k, and that the proof is cut-free.

$$\dfrac{\begin{array}{ccccccc} & & & & A(0) & & \\ & & & & \vdots\ kth\ subtree & & \\ & & A(0) & & A(\underline{k-1}) & & \\ \vdots\ P(0) & & \vdots\ P(1) & & \vdots\ P(k) & & \\ A(0) & & A(\underline{1}) & \ldots & A(\underline{k}) & \ldots & \end{array}}{\forall x\ A(x)}\ constructive\ \omega\ rule$$

Figure 8–2: Form of Linear Proof in $PA_{c\omega}$

As discussed in Section 7.2, induction is taken to be a derived rule in $PA_{c\omega}$. So, if the cut rule is not used explicitly, use of such a rule will take the form shown in Figure 7–1. However, the proof given in this figure is not the final form of the proof without the use of cuts, and even as it stands there are other leaves present in the tree which prevent Figure 7–1 from having the linear structure of Figure 8–2. It is the aim of this section to show that use of induction does however correspond to a linear form, in the sense that a linear general proof will correspond to some standard inductive proof (and vice versa).

Incidentally, the structure of an ω-rule proof may also take what I shall call a step linear form, in which different subtrees may be developed in different ways, yet with there still being a recognisable algorithm for their construction; for example:

$$\dfrac{\begin{array}{cccccccc} & & & & \vdots\ P(0) & & \vdots\ P(1) & \\ & & & & A(0) & & A(\underline{1}) & \\ \vdots\ P(0) & & \vdots\ P(1) & & \vdots\ P(2) & & \vdots\ P(3) & \\ A(0) & & A(\underline{1}) & & A(\underline{2}) & & A(\underline{3}) & \ldots \end{array}}{\forall x A(x)}\ \omega\ rule$$

Note that the splitting of the general proof (ie. branching via rewrite rules) corresponds to step linearity; in practice this does not happen very often, but an

example of this type would be $\forall x(even(x) \vee even(s(x)))$. No examples which split in this way are in fact considered in Appendix E.8.3 (in which the transcripts of the implementational system running are provided), although the implementational system is designed to allow such splits; thus, these general proof examples have a "linear" shape in one sense within the implementational tree representation, but this should not be confused with the criterion for linearity which is presented above and which is what shall be meant by "linear" in this thesis.

The nested case, in which there is more than one use of the ω-rule, is discussed further in Section 9.6. Use of two inductions should be compared with P. Madden's transformations which linearise proofs defined by functions with two inductive variables (see Section 11.2) (Madden, 1989). In general, of course, any number of inductions could be used.

8.3.2 Linearity and General Proofs

As discussed in Chapter 6, a general proof is a parametrised version of an arbitrary subtree of an ω-proof, which is a proof in $PA_{c\omega}$ of a universal statement. Thus, a "linear" general proof will correspond to the kth subtree of Figure 8–2, which will have the form $A(k)$ reducing in a uniform manner to $A(k-1)$ to $A(k-2)$ down to $A(0)$.

I wish to describe a procedure for standard induction proofs which yields ω-proofs, and to characterise proofs obtained by carrying out this transformation; by these means, if it is possible to put a general proof in a linear form, then one could obtain an inductive proof in PA by inverting this process. Analysis of this process would suggest the provision of a suitable cut formula. I shall show in this subsection that if $\forall x A(x)$ is provable by induction in $PA_{c\omega}$, then a proof is of the form given in Figure 8–2, and thus that this corresponds to strong linearity.

Note that the general proofs considered are a special case of ω-proofs, in that in practice manipulations are just being carried out on a single formula (cf. use of rewrite rules in the general proof), as opposed to arbitrary sequent manipulations.

Hence, the linearisation of the general-proof is its transformation, via the definition of a formula Q, to a proof which has the structure in the mth subtree of

$$\dfrac{\dfrac{Q(m)}{Q(m-1)}\,i}{\begin{array}{c}\vdots\, i's\\ Q(0)\end{array}}$$

where the proof of $Q(m)$ is an extension of the proof of $Q(m-1)$, by the proof steps (via rewrite rule block) i. For the step version of linearity, Q would be different in different subtrees. So long as there is not a case-split in the ω-proof, the notion of linearity considered will be the strong one.

I shall show that the linear shape, and "inductive" structure of such a proof would suggest that induction could be used on it, and hence that the original proof might be completed using a cut with cut-formula $\forall m Q(m)$, and then induction.

The idea is to turn the general proofs into a linear form, which will suggest that induction can be done on the original formula (that is to say that this formula is a suitable cut formula). Hence it is necessary to recognise Q such that the following may be transformed:

Original General Proof:

$$\begin{array}{c}P(s(n))\\ \vdots\\ EQUALITY\end{array}$$

$$\Rightarrow\ by\ transformation$$

into

New General Proof:

$$\dfrac{\dfrac{\dfrac{Q(s(n))}{Q(n)}\,i}{\begin{array}{c}\vdots\\ Q(0)\end{array}}}{EQUALITY}\,j$$

These correspond to the proof in PA, where $\forall x\ P(x)$ was the original formula:[4]

---

[4]As mentioned above, one is concerned for the stages i and j with application of rewrite rules on a formula, rather than structural manipulations on a sequent.

$$\dfrac{\forall x\, Q(x) \vdash \forall x P(x) \qquad \dfrac{\begin{array}{cc} CLOSES & CLOSES \\ \vdots j & \vdots i \\ \vdash Q(0) & Q(r) \vdash Q(s(r)) \end{array}}{\vdash \forall x\, Q(x)}\, ind(x)}{\vdash \forall x\, P(x)}\, cut$$

Given such an inductive proof, would it be possible to generate a proof in $PA_{c\omega}$ (ie. an ω-proof)? In order to do this, one needs Theorem 8.1. The fact that if a proof of $\Gamma, A(\underline{k}) \vdash A(s(\underline{k}))$ is provided, then there is a derivation of $\Gamma \vdash A(s(\underline{k}))$ from $\Gamma \vdash A(\underline{k})$ (in the sense of there being a natural deduction proof of the former from the latter, possibly using additional axioms) forms an analogue of the deduction theorem.[5]

Theorem 8.1 (Deduction Theorem for Sequents) *The sequent $\Gamma, A \vdash B$ is provable in $PA_{c\omega}$ if and only if there is a derivation in $PA_{c\omega}$ of*

$$\begin{array}{c} \Gamma \vdash A \\ \vdots \\ \Gamma \vdash B \end{array}$$

Proof of Theorem 8.1:

$\Rightarrow$: Suppose $\Gamma, A \vdash B$ is provable in $PA_{c\omega}$. The cut rule may be used to derive $\Gamma \vdash B$ from $\Gamma \vdash A$ and $\Gamma, A \vdash B$. Hence, given $\Gamma, A \vdash B$, then there is a derivation of $\Gamma \vdash B$ from $\Gamma \vdash A$. By the cut elimination theorem, there must be a cut-free derivation (in $PA_{c\omega}$) of $\Gamma \vdash B$ from $\Gamma \vdash A$.

$\Leftarrow$: Suppose that there is a proof in $PA_{c\omega}$ of $\Gamma \vdash B$ from $\Gamma \vdash A$. By using the same rules, there must be a proof of $\Gamma, A \vdash B$ from $\Gamma, A \vdash A$. However, $\Gamma, A \vdash A$ is an axiom. Hence, there is a proof in $PA_{c\omega}$ of $\Gamma, A \vdash B$.

Theorem 8.2 (Linearisation Theorem) *Q can be proved using a single induction if and only if there is a strongly linear ω-proof of Q*

---

[5] If $\Gamma, B \vdash_K A$, then $\Gamma \vdash_K B \rightarrow A$, for some logical system K (Dummett, 1977, P127).

Proof of Theorem 8.2:

Induction ⇒ Linear General Proof

By means of Theorem 8.1, the fact that induction carries through for Q will imply directly that the general proof is linear. So, if $Q(x)$ is such that induction can be done, ie. $Q(s(x))$ reduces to $Q(x)$, by the rules i, and $\vdash Q(0)$, then the general proof will be of the form:

$$\begin{array}{c} Q(s^n(0)) \\ \text{reduces to} \\ \Downarrow \\ \text{by rules } i(n) \\ Q(s^{n-1}(0)) \\ \vdots \\ Q(0) \\ \text{which holds} \end{array}$$

The rules $i(k)$ here form a repeated (parametrised) rule-block. This is what has been described above as a linearisable general proof. So, for arithmetic, if Q is suitable for induction, then the general proof is linearisable.

Incidentally, the general proof may then be used as the kth (for arbitrary k) subtree of the ω-rule, to give a linear ω-proof of $\forall x Q(x)$, and also such that if induction carries through for Q, then there is a suitable cut formula provided for the proof in $PA_{c\omega}$, too, as follows (cf. discussion in Subsection 7.2.1):

$$\dfrac{\forall x Q(x) \vdash \forall x P(x) \qquad \dfrac{\vdash Q(0) \quad \dfrac{\vdash Q(0)}{\vdash Q(1)} \quad \ldots \quad \begin{array}{c}\vdash Q(0) \\ \vdots \\ \vdash Q(\underline{k}) \\ \textit{kth subtree}\end{array} \quad \ldots}{\vdash \forall x\, Q(x)}\ \omega\ rule}{\vdash \forall x P(x)}\ cut$$

Linear General Proof ⇒ Induction

Does the converse hold, that is, does a linear form of the new general proof imply that induction may be used directly on Q, such that Q would be an appropriate

cut formula? Given a general proof (cf. representation of an arbitrary subtree of the ω-proof) of the form

$$\dfrac{\dfrac{Q(s(k))}{Q(k)}\,i(k)}{\begin{array}{c}\vdots\\ Q(0)\end{array}}\,j$$
$$\overline{\qquad equality\qquad}$$

then $Q(s(\underline{k}))$ reduces to $Q(\underline{k})$ for all numerals such that $k \leq n$, where n was arbitrary — hence k is arbitrary. (The rules $i(k)$ here form a repeated rule-block (parametrised over k): this is what has been described above as a linearisable general proof.) By the soundness of the general proof representation with respect to $PA_{c\omega}$ (cf. Section 6.4), there is a proof in $PA_{c\omega}$ of

$$\vdash Q(\underline{k})$$
$$\vdots$$
$$\vdash Q(s(\underline{k}))$$

and by Theorem 8.1, there is a proof of $Q(k) \vdash Q(s(k))$, for each numeral k. By the form of the original strongly linear proof, each proof segment $P(k)$ is obtained by taking the same shape of proof with a numerical parameter that is replaced by the appropriate value of k. So, there is a uniform correspondence between how the different instances of the numeral k are treated in each proof of $Q(k) \vdash Q(s(k))$, which suggests that the free variable version $Q(r) \vdash Q(s(r))$ is provable. The final lines of the general proof provide a proof of $Q(0)$, and therefore the prooftree in PA using induction may be completed.

Hence, **general proof of Q strongly linear $\equiv$ induction will succeed on Q.**

8.3.3 Illustration with Respect to a General Proof

In this subsection the discussion above shall be related to an actual general proof, with the aim of demonstrating how it is necessary to generalise in order to get a linear shape. The (meta)inductions involved in the correctness of rewriting within the general proof will be highlighted, to show how the variable candidates for generalisation apart are revealed by such meta-inductions, and thus also the induction that appears from the linear general proof.

	$(\underline{n}+\underline{n})+\underline{n}=\underline{n}+(\underline{n}+\underline{n})$
$\underline{n} \equiv s^n(0)$	$(\underline{s^n(0)+s^n(0)})+s^n(0)=\underline{s^n(0)+(s^n(0)+s^n(0))}$
USE 6.2 *n TIMES BOTH SIDES*	$\underline{s^n(0+s^n(0))}+s^n(0)=\underline{s^n(0+(s^n(0)+s^n(0)))}$
USE 6.1 *BOTH SIDES*	$\underline{s^n(s^n(0))+s^n(0)}=\underline{s^n(s^n(0)+s^n(0))}$
USE 6.2 *n TIMES ON LEFT*	$s^n(s^n(0)+s^n(0))=s^n(s^n(0)+s^n(0))$
	EQUALITY

Figure 8–3: Another General Proof of $\forall x\ (x+x)+x=x+(x+x)$

Figure 8–3 forms a counterpart to Figure 6–1, in which the order of the applications of rewrite rules is the same, but the blocks of application of individual rewrite rules have been underlined for clarification. For this example, it is the case that (for $0 \leq i \leq n$) the first step of the general proof given in Figure 8–3 may be represented by:

$$Q(i) \equiv s^i((s^{n-i}(0)+s^n(0))+s^n(0)=s^i(s^{n-i}(0)+(s^n(0)+s^n(0)))$$

Induction on i (requiring proof of the step case $Q(i) \vdash Q(s(i))$) is proved by an application of 6.2 on each side (ie. $s(X)+Y \Rightarrow s(X+Y)$). $Q(0)$ is the original sequent of the general proof.

$$Q(n) \equiv s^n(0+s^n(0))+s^n(0)=s^n(0+(s^n(0)+s^n(0)))$$

which is the next line of the general proof, obtained after n such individual applications of 6.2 on each side. 6.1 is then used once on each side of the general proof to remove surplus zeros. Finally, the conversion

$$s^n(X)+s^n(0) \Rightarrow s^n(X+s^n(0))$$

is used to alter the left hand side. Each intermediate line of the general proof will be of the following form (after j applications of 6.2), and (linear) induction (for $0 \leq j \leq n$) may be used to justify this

$$Q'(j) \equiv s^j((s^{n-j}(X))+s^n(0))=\ldots$$

Combining Q and Q', such that each rewriting step may be made individually, at any appropriate stage, one obtains the general form of a line within the general proof as being of the form

$$Q''(i,j) \equiv s^j(s^{i-j}(s^{n-i}(0)+s^n(0))=s^i(s^{n-i}(0)+(s^n(0)+s^n(0))\ (*)$$

This does not simplify. Such a Q'' representation allows many different orders of proof lines, and hence represents several shapes of proofs, all of which will result in a correct general proof. It is possible to use induction on i and j (provable by use of 6.2, and dealing with first i and then j, for instance) to get from $Q''(0,0)$ to $Q''(n,n)$.

$$Q''(0,0) \equiv (s^n(0) + s^n(0)) + s^n(0) = s^n(0) + (s^n(0) + s^n(0))$$

$$Q''(n,n) \equiv s^n((0 + s^n(0)) + s^n(0)) = s^n(0 + (s^n(0) + s^n(0)))$$

$Q''(n,n)$ may be proved by using 6.1 (ie. $0 + Y \Rightarrow Y$) twice.

By looking at the structure of Q'', one can see that induction is taking place on the first and fourth terms only, and this suggests that the generalisation should involve replacing these terms by a new variable, and doing induction on this, ie. $\forall y\ (y + x) + x = y + (x + x)$. Note that in this example the linear nature of the general proof is being made explicit, because the repeated blocks from $Q(s(n))$ to $Q(n)$ are not present without reordering (ie. the unreordered proof is *not* linear). Other examples will involve an even more obvious linearisation of the proof. The structure given by $(*)$ in the previous paragraph is in fact here just formalising the process occurring in the general proof, by which it may be seen that certain of the terms in the proof remain unaltered. This will not occur in many proofs, notably the list examples.

The way that the linearisation process ties up with the general proofs is that, from the structure of the rewrite rules, it is apparent that terms in certain argument positions must remain unaltered. This may be seen from the form of Q'', for example. Note that from an initial inspection of the formula $(x_1 + x_2) + x_3 = x_4 + (x_5 + x_6)$, it is not necessarily possible to deduce that x_3 and x_6 cannot be altered using just the recursive definitions of addition, for x_2 might be altered if x_1 could be made to 'disappear', because then it would be in the definitional argument position, and similarly x_3 could be altered if x_1 and x_2 could be made to disappear. So, another approach would be preferable. The general proof shows that it is not necessary to achieve equality, ie. one does not *have* to alter x_2, although it is possible to do so. Similarly, from the general proof, one concludes

that x_5 may remain unaltered. So there does exist more information than would be provided by an initial guess. The explanation-based generalisation (EBG) proof, of course, must express this information too, ie. it must express that x_3 and x_6 are distinct from, say x_1 and x_4, and also that x_2 and x_5 are too (since they are unaltered in the general proof, and the same rules are used). What the EBG proof does is to provide a most general generalisation of the initial formula which can be proved using the rules given in the general proof. However, in some cases (and especially in the list examples), such a generalisation will be the same, and still not provable by induction, since what is needed is the addition of new structure (ie. an 'accumulator').

Next, the case of the general proof of the generalisation obtained from the EBG method is considered. This proof is linear. Define

$$Q(n) \equiv (s^n(0) + s^m(0)) + s^k(0) = s^n(0) + (s^m(0) + s^k(0))$$

Then $Q(s(n))$ reduces to $Q(n)$, for all n:

$$(s(s^n(0)) + s^m(0)) + s^k(0) = s(s^n(0)) + (s^m(0) + s^k(0))$$

$$\Rightarrow s(s^n(0) + s^m(0)) + s^k(0) = s(s^n(0) + (s^m(0) + s^k(0)))$$

$$\Rightarrow s((s^n(0) + s^m(0)) + s^k(0)) = \ldots$$

$$\Rightarrow EQUALITY \text{ using } Q(n)$$

$$Q(0) \equiv (0 + s^m(0)) + s^k(0) = 0 + (s^m(0) + s^k(0))$$

which is provable by two application of 6.1. Thus the general proof is of the form:

$$\dfrac{\dfrac{\dfrac{Q(n)}{Q(n-1)}\, i}{\vdots\, i}}{\dfrac{Q(0)}{EQUALITY}\, j}$$

Note the connection between this linearity and induction on the generalisation obtained, and how it is possible to see that this generalisation will be provable by induction, unlike the original formula. The (internal proof) structure corresponds to Q'', described above in the linearisation of the general proof, since the rules

are the same. However, rather than directly suggesting what the first and fourth terms should be, Q'' really tells us that inductive processes are taking place on the first and fourth terms in the proof, and so these are the candidates for induction. Note that the general proof of the generalisation is the EBG proof with the final (instantiated) result. Thus, the EBG proof will necessarily be linear, if it provides a correct cut formula, ie. if the top sequent is provable by induction. By doing EBG, we rename the parts which should not be altered when using induction. This puts the general proof into a linear form, in that (for inital line $P(n)$) $P(n)$ reduces to $P(n-1)$, whereas it did not before. By the linearisation argument, this must be suitable for induction. See Subsection 8.5.1, where $Q(n, n, n)$ reduces to $Q(n-1, n, n)$, where the second and third arguments are renamed by the EGB process. Hence, if a solution is possible by generalisation of variables apart, then this method will find it (so long as a correct general proof is given initially!). Thus, EBG is a way of linearising the general proof. See the following section for a development of this approach, with respect to lists.

8.4 Linearisation for List Examples

A more detailed examination of linearisation for the list examples is presented in this section. These are discussed with reference to the different categories of generalisation presented in Section 3.1, namely generalisation of terms to variables, generalisation apart and generalisation of a constant in an accumulator position to a variable.

In Appendix C a summary is provided of the results related to the various of the more interesting list examples already considered, and also it is shown that the generalisations suggested really are suitable cut formulae in the sense that ordinary induction may be performed upon them.

8.4.1 Generalisation Apart:

$$\forall l.\ (l <> l) <> l = l <> (l <> l)$$

In this subsection the corresponding list example to the arithmetical example $\forall x\ (x + x) + x = x + (x + x)$ $(*)$ given in Section 8.1 is considered, namely $\forall l.\ (l <> l) <> l = l <> (l <> l)$ (where $<>$ denotes list concatenation). The following rewrite rules are used:

$$nil <> X \Rightarrow X \tag{8.13}$$

$$(H :: T) <> X \Rightarrow H :: (T <> X) \tag{8.14}$$

The general proof is as follows:

$$\begin{array}{rcl}
(\mathcal{L}(n) <> \mathcal{L}(n)) <> \mathcal{L}(n) & = & \mathcal{L}(n) <> (\mathcal{L}(n) <> \mathcal{L}(n)) \\
((x_{s^{n-1}(0)} :: \mathcal{L}(n-1)) <> \mathcal{L}(n)) <> \mathcal{L}(n) & = & (x_{s^{n-1}(0)} :: \mathcal{L}(n-1)) <> (\mathcal{L}(n) <> \mathcal{L}(n)) \\
 & & \textit{by def of } \mathcal{L}(n) \\
(x_{s^{n-1}(0)} :: (\mathcal{L}(n-1) <> \mathcal{L}(n))) <> \mathcal{L}(n) & = & x_{s^{n-1}(0)} :: (\mathcal{L}(n-1) <> (\mathcal{L}(n) <> \mathcal{L}(n))) \\
 & & \textit{by } 8.14 \\
x_{s^{n-1}(0)} :: ((\mathcal{L}(n-1) <> \mathcal{L}(n)) <> \mathcal{L}(n)) & = & x_{s^{n-1}(0)} :: (\mathcal{L}(n-1) <> (\mathcal{L}(n) <> \mathcal{L}(n))) \\
 & & \textit{by } 8.14 \\
 & \vdots & \textit{similarly, } n-1 \textit{ times} \\
x^{s^{n-1}(0)} :: \ldots :: (nil <> \mathcal{L}(n)) <> \mathcal{L}(n) & = & x^{s^{n-1}(0)} :: \ldots :: (nil <> (\mathcal{L}(n) <> \mathcal{L}(n))) \\
 & & \\
x^{s^{n-1}(0)} :: \ldots :: (\mathcal{L}(n) <> \mathcal{L}(n)) & = & x^{s^{n-1}(0)} :: \ldots :: (\mathcal{L}(n) <> \mathcal{L}(n)) \\
 & & \textit{EQUALITY}
\end{array}$$

Compare this with the (general) proof for $\forall x\ (x + x) + x = x + (x + x)\ x \in \mathbb{N}$ given in Figure 6–1. Just as $s^n(0)$ remains unchanged in the 2nd, 3rd, 5th and 6th places, $\mathcal{L}(n)$ remains unchanged in these positions too. So, one could generalise to $(l <> p) <> p = l <> (p <> p)$ and so induction on l, in a similar manner. For simplification the proposed generalisation heuristic is being used here in order to prove a generalisation to enable the proof to be completed using the cut rule, rather than the most general generalisation, which could also be extracted from the proof by the explanation-based generalisation method described in Section 8.1.4 and written as $\forall l \forall p \forall q\ (l <> p) <> q = l <> (p <> q)$ (see the following page).

There is also an analogy to the use of "general" rules such as $s^n(0)+Y \Rightarrow s^n(0+Y)$ from $s(X)+Y \Rightarrow s(X+Y)$, namely rules like $l(s(n)) <> m \Rightarrow x_{s^n(0)} :: (x_{s^{n-1}(0)} :: \ldots :: (x_0 :: m)))$ from $x_{s^n(0)} :: l(n) <> m \Rightarrow x_{s^n(0)} :: (l(n) <> m)$.

This example is not unique: any examples of this type will carry through in a similar manner; for instance, $\forall l.\ l <> (a :: l) = (a :: l) <> l$, where :: denotes appending an element to a list (note that this is actually a non-theorem), compares with "$x+s(x) = s(x)+x$" — it would be directly comparable if $[a, b] \equiv [b, a]$. The second l in the general proof on each side is left unchanged, so the generalisation $l <> (a :: p) = (a :: l) <> p$, with induction on l, would be guessed if list order were not important, ie. if the general proof resulted in equality.

Both these cases are directly analogous to the natural numbers case, and the heuristic is to see what remains unchanged throughout the general proof and replace this by a new variable (on which induction is not carried out). Hence, a heuristic for this case is:
"try replacing l by p if L(n) is obviously left unaltered in the general proof, and doing induction on l"
where "left unaltered" is taken in the same sense as for the heuristic for arithmetic mentioned earlier — that it is not broken down in the general proof.

Explanation-Based Generalisation Proof

EBG Proof

rule	general		proof	instantiation
$8.14([2], n)$	W	$=$	$(y_1 :: \ldots y_n :: P) <> X$	
$8.14([1,1], n)$	W	$=$	$y_1 :: \ldots y_n :: (P <> X)$	
$8.14([1], n)$	$fn(z_1 :: \ldots :: z_n :: (Q <> Y) \mid Z)$	$=$	$y_1 :: \ldots y_n :: (P <> X)$	δ
$6.2([2,2,1,1], 1)$	$z_1 :: \ldots z_n :: ((Q <> Y) <> Z)$	$=$	$y_1 :: \ldots y_n :: (P <> X)$	$fn \equiv <>$
$6.2([2,2,2], 1)$	$z_1 :: \ldots z_n :: (Y <> Z)$	$=$	$y_1 :: \ldots y_n :: (P <> X)$	$Q = nil$
$EQUALITY$	$z_1 :: \ldots :: z_n :: (Y <> Z)$	$=$	$y_1 :: \ldots y_n :: X$	$P = nil$
	X	$=$	$Y <> Z$	$y_i = z_i$

$\delta : W = fn(((z_1 :: \ldots :: z_n :: Q) <> Y) \mid Z)$

Filtering back, $((z_1 :: \ldots :: z_n :: nil) <> Y) <> Z = (z_1 :: \ldots z_n :: nil) <> (Y <> Z)$ is obtained. This suggests the cut formula of $\forall l\ (l <> y) <> z = l <> (y <> z)$ (C). This method works for lists, in an analogous way to the provision of an EBG proof for $(*)$. However, this cut formula does not involve accumulators — for these examples the EBG proof cannot produce the cut formula since the given formula is not an instance of the "generalised" one. Induction on (C) carries through by weak fertilisation, in an analogous way to the $(*)$ example, as follows:

$(T <> T2) <> T3 = T <> (T2 <> T3)$

$\vdash (H :: T <> T2) <> T3 = H :: T <> (T2 <> T3)$

$\ldots \vdash H :: ((T <> T2) <> T3) = H :: (T <> (T2 <> T3))$

Hence, explanation-based generalisation will again produce a correct most general generalisation in an analogous manner to the explanation of Subsection 8.1.4.

Linearisation of the General Proof

This may be carried out by noting, in the case of lists, that

$$(\mathcal{L}(m) <> \mathcal{L}(n)) <> \mathcal{L}(n) = \mathcal{L}(m) <> (\mathcal{L}(n) <> \mathcal{L}(n))$$

reduces via the general proof to:

$$x^{m-1}(0) :: (\mathcal{L}(m-1) <> \mathcal{L}(m)) <> \mathcal{L}(n) = x^{m-1}(0) :: (\mathcal{L}(m-1) <> (\mathcal{L}(n) <> \mathcal{L}(n)))$$

and hence, by defining

$$Q(m) = (\mathcal{L}(m) <> \mathcal{L}(n)) <> \mathcal{L}(n) = \mathcal{L}(m) <> (\mathcal{L}(n) <> \mathcal{L}(n))$$

one may obtain the general proof structure

$$\begin{array}{c} Q(n) \\ \hline Q(n-1) \end{array} i$$
$$\vdots$$
$$Q(0)$$

This suggests the generalisation $\forall l\ (l <> p) <> p = l <> (p <> p)$. (Note that the EBG method above will give the most general generalisation of this type, although the above would suffice). Alternatively, the final line of the general proof, namely $x_{s^{n-1}(0)} :: \ldots :: x_{s(0)} :: (nil <> \mathcal{L}(n)) <> \mathcal{L}(n) = x_{s^{n-1}(0)} :: \ldots :: x_{s(0)} :: (nil <> (\mathcal{L}(n) <> \mathcal{L}(n)))$ could have been generalised with respect to $\mathcal{L}(0)$, to give Q.

General Proof of Generalisation

The generalisation under consideration is (C) above. The general proof will reach equality using the same rules as in the general proof of $(*)$. This proof is linear.

Note that this corresponds exactly to induction, where it is necessary to show that $Q(k) \vdash Q(s(k))$, and $\vdash Q(0)$. In other words, both methods equate to showing that the proof of $Q(s(k))$ reduces to the proof of $Q(k)$ by certain rewrite rules. In this case, $Q(n) \equiv (\mathcal{L}(n) + \mathcal{L}(m)) + \mathcal{L}(k) = \mathcal{L}(n) + (\mathcal{L}(m) + \mathcal{L}(k))$ rewrites to $x_{s^{n-1}(0)} :: Q(n-1)$, in an analogous manner to that shown in the previous subsection.

Rule Permutation

However, there is also the case of rule permutation to be considered. In fact the order of the rules used may be rearranged, so that the structure of the first and fourth $\mathcal{L}(n)$ is expanded totally at the outset, and then the same rule applied as many times as possible. Hence, one may apply $\underbrace{R_1R_2R_3}_{\textit{block one}} \ldots \underbrace{R_1R_2R_3}_{\textit{block N}}$, or, equivalently, $\underbrace{R_1 \ldots R_1}_{N \textit{ times}} \underbrace{R_2 \ldots R_2}_{N \textit{ times}} \underbrace{R_3 \ldots R_3}_{N \textit{ times}}$. Compare this with the general proof of $(*)$, given in Figure 6–1. Just as $s^n(0)$ remains unchanged in the second, third, fifth and sixth places, $\mathcal{L}(n)$ remains unchanged in these positions too. Note also the correspondence between the rules in both general proofs, the rules written in the order corresponding to the second representation above, ie. $8.14([2], n)$, $8.14([1,1], n)$, $8.14([1], n)$, $8.13(P1)$, $8.13(P2)$ directly correspond to $6.2([2], n)$, $6.2([1,1], n)$, $6.2([1], n)$, $6.1(P1)$, $6.1(P2)$, the latter being one ordering of the rules in the general proof of $(*)$ (although several are possible).[6]

8.4.2 Generalisation by Adding Accumulators: $\forall l.\ rotate(len(l), l) = l$

This example requires the use of new methods, since extra structure is introduced into the general proof, and it is no longer possible to use the heuristic proposed in

---

[6]For comparison, recall that $8.14 : (H :: T) <> X \Rightarrow H :: (T <> X)$;
$6.2 : s(X) + Y \Rightarrow s(X + Y)$; $8.13 : nil <> X \Rightarrow X$; $6.1 : 0 + Y \Rightarrow Y$

Section 8.1, nor the explanation-based generalisation method of which the heuristic is a simplification.

If it is not the case that there are $\mathcal{L}(n)$ left unchanged throughout the proof, it might however be possible to glean some information. For example, if the original proof is $P(m)$, and this can be seen to reduce to $P(m-1)$ in some way, this would suggest that induction might be used. This example shall be considered in order to see what is happening in the general proofs and whether a general pattern is seen to emerge; it is also considered in Appendix C.2, in which proof by induction is shown to be blocked, whereas induction is shown to carry through for the suggested generalisation.

It is necessary to make use of the following rewrite rules (in addition to those given above):

$$rotate(0, L) \Rightarrow L \tag{8.15}$$

$$rotate(s(n), H :: T) \Rightarrow rotate(n, T <> (H :: nil)) \tag{8.16}$$

$$len(nil) \Rightarrow 0 \tag{8.17}$$

$$len(H :: T) \Rightarrow s(len(T)) \tag{8.18}$$

The general proof is as follows:

$$\begin{array}{rcl}
rotate(s(len(\mathcal{L}(n-1))), \mathcal{L}(n)) & = & \mathcal{L}(n) \\
rotate(len(\mathcal{L}(n-1)), x_{s^{n-1}(0)} :: \mathcal{L}(n-1)) & = & \mathcal{L}(n) \\
 & & \textit{by len, } \mathcal{L} \textit{ defs} \\
rotate(len(\mathcal{L}(n-1)), \mathcal{L}(n-1) <> (x_{s^{n-1}(0)} :: nil)) & = & x_{s^{n-1}(0)} :: \mathcal{L}(n-1) \\
 & & \textit{by } 8.16 \\
rotate(len(\mathcal{L}(n-1)), (x_{s^{n-2}(0)} :: \mathcal{L}(n-2)) <> (x_{s^{n-1}(0)} :: nil)) & = & x_{s^{n-1}(0)} :: \mathcal{L}(n-1) \\
 & & \textit{by } \mathcal{L} \textit{ def} \\
rotate(s(len(\mathcal{L}(n-2))), x_{s^{n-2}(0)} :: (\mathcal{L}(n-2) <> (x_{s^{n-1}(0)} :: nil)) & = & x_{s^{n-1}(0)} :: \mathcal{L}(n-1) \\
 & & \textit{by } 8.14 \\
rotate(len(\mathcal{L}(n-2)), (\mathcal{L}(n-2) <> x_{s^{n-1}(0)} :: nil) <> (x_{s^{n-2}(0)} :: nil)) & = & x_{s^{n-1}(0)} :: x_{s^{n-2}(0)} :: \mathcal{L}(n-2) \\
 & & \textit{by } 8.16, \mathcal{L} \textit{ def} \\
 & \vdots & \textit{similarly, } n \textit{ times} \\
rotate(len(\mathcal{L}(0)), (\mathcal{L}(0) <> x_{s^{n-1}(0)} :: nil) <> \ldots <> x_0 :: nil) & = & x_{s^{n-1}(0)} :: \ldots :: x_0 :: nil \\
x_{s^{n-1}(0)} :: nil <> \ldots <> x_0 :: nil & = & x_{s^{n-1}(0)} \ldots x_0 :: nil
\end{array}$$

Is there a general pattern in the proof which would suggest a generalisation? For the mth case of the above proof, such that $m < n$,

$$rotate(len(\mathcal{L}(m)), (\ldots(\mathcal{L}(m) <> x_{s^n(0)}) <> \ldots) <> x_{s^m(0)} :: nil))$$
$$= rotate(len(\mathcal{L}(m)), \mathcal{L}(m) <> (x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: nil))$$

$= x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: nil <> \mathcal{L}(m)$

$= x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: \mathcal{L}(m)$

Let $a_m = x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: nil$.

Then $rotate(len(\mathcal{L}(m)), \mathcal{L}(m) <> a_m) = a_m <> \mathcal{L}(m)$ is a suitable generalisation, the right hand side of which is produced by setting $F(x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: nil, \mathcal{L}(m)) = x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: \mathcal{L}(m)$ and solving for F.

In fact, it is only necessary to note that

$$R(s(n)) \equiv rotate(len(\mathcal{L}(s(n))), \mathcal{L}(s(n))) = \mathcal{L}(s(n))$$

reduces to

$$R'(n) \equiv rotate(len(\mathcal{L}(n)), \mathcal{L}(n) <> (x_{s^n(0)} :: nil)) = x_{s^n(0)} :: \mathcal{L}(n)$$

$$= (x_{s^n(0)} :: nil) <> \mathcal{L}(n)$$

and compare R with R' to arrive at the generalisation

$$rotate(len(l), l <> a) = a <> l.$$

Note that in fact the $x_{s^n(0)} :: nil$ on the left hand side of R' has caused the goal of writing the right hand side as some $F(x_{s^n(0)} :: nil)$.

In both cases, the step of getting from $x :: list$ to $(x :: nil) <> list$ is a key step which is not obvious. However, it can be seen from the proof that what is needed is to rewrite $x :: list$ into some form involving $x :: nil$, in order to rewrite the same terms on either side. The provision of any expression which was equivalent to $x :: list$ involving $x :: nil$ would suffice, and this stage could be carried out even mechanically, by using various stored rewrite rules, together with pattern matching. However, this process could be regarded in general as open-ended, and provides a topic for further work.

Generalising from $Q(0)$

The general method suggested in this section has been to consider $R(n)$ and $R'(n-1)$: various lines in the general proof, where $R(n)$ reduced to $R'(n-1)$, and both

were at their most structurally similar state. In other words, $R(n)$ reduced to $R'(n-1)$ using i steps, just as $R(n-1)$ would reduce to $R'(n-2)$ using the same i steps. A heuristic would be to leave structure common to R and R' the same, do induction on l if $\mathcal{L}(n)$ is at the same structural point in $R(n)$ as $\mathcal{L}(n-1)$ in $R'(n-1)$, and to generalise any other common subexpressions in R'. I shall now consider in more depth the suggestion that instead $R'^{\cdots\prime}(0)$ is the appropriate object of generalisation, primarily because some of the term rewriting stages to equality would only be done at the end of the general proof, but these might be needed to provide equal generalisation candidates on both sides of the equation, rather than $R(n)$ being compared with $R'(n-1)$. So, this example will be considered with reference to this suggestion.

In this case,

$Q(0) \equiv rotate(len((\mathcal{L}(0))), (\mathcal{L}(0) <> x_{s^n(0)} :: nil) <> \ldots <> x_0 :: nil)$

$= x_{s^n(0)} :: \ldots :: x_0 :: \mathcal{L}(0)$

This rewrites to (using 'new' rewrites):

$rotate(len(\mathcal{L}(0)), \mathcal{L}(0) <> (x_{s^n(0)} :: \ldots :: x_0 :: nil))$

$= x_{s^n(0)} :: \ldots :: x_0 :: \mathcal{L}(0)$

$= x_{s^n(0)} :: \ldots :: x_0 :: nil <> \mathcal{L}(0)$

The generalisation suggested is, as before,

$$rotate(len(l), l <> a) = a <> l$$

Hence, in this case there is no advantage to one approach over the other.

Rule Permutation

In this subsection the same example is considered, with relation to the correspondence between linearisation and permutation of rules within the general proof.

As seen above, the rules of the general proof are

$$\underbrace{(8.16 + 8.14)\ \text{n-i times} + 8.13\ \text{once}}_{\textit{repeat } n-1 \textit{ times}} + 8.15$$

Equivalently,

$$\underbrace{8.16 + 8.14 + 8.16 + 8.14 + \ldots}_{repeat} + \overbrace{\underbrace{8.14 + 8.13}_{repeat} + 8.15}^{can\ swap}$$

There are not any other possibilities, using the given rewrite rules.

The normal EBG proof, using the same rules as in the general proof and filtering back, suggests $rotate(s^n(0), H :: z_1 :: \ldots :: z_{n-1} :: nil) = H :: z_1 :: \ldots z_{n-1} :: nil$, which is not a correct cut formula. This is because the generalisation required involves accumulators, and hence will not use this sequence of rules.

Induction on the original equation is blocked (although the general proof essentially allows a continuation of this blocked proof). Induction on $\forall al\ rotate(len(l), l <> a) = a <> l$ works, with one induction, plus use of the axiom $(X <> Y) <> Z = X <> (Y <> Z)$. (See Appendix C.2 for details.)

Upon attempting to linearise the general proof, it will be found that $P(s(n))$ does not reduce to $P(n)$:
$P(s(n)) \equiv rotate(len(\mathcal{L}(s(n))), \mathcal{L}(s(n)) = \mathcal{L}(s(n))$ (1st line of GP)
$\equiv rotate(s^n(0), x_{s^n(0)} :: \mathcal{L}(n)) = x_{s^n(0)} :: \mathcal{L}(n)$
$\equiv rotate(len(\mathcal{L}(n)), \mathcal{L}(n) <> (x_{s^n(0)} :: nil)) = x_{s^n(0)} :: \mathcal{L}(n)$
which is not equivalent to $P(n)$.

The general proof of the generalisation $rotate(len(\mathcal{L}(n)), \mathcal{L}(n) <> a) = a <> \mathcal{L}(n)$ shall now be considered. This uses different rules to the general proof of the original equation, and hence the explanation-based generalisation method on the original goal will not produce this generalisation. The rules in the general proof will not apply immediately to this case, since the second argument of *rotate* is not of the form $H :: T$. As shown below, the same rules are present, but in a different order, plus final justifications.

$$
\begin{array}{rcll}
rotate(s^n(0), (x_{s^{n-1}(0)} :: \ldots x_0 :: nil) <> a) & = & a <> \mathcal{L}(n) & \\
 & & & 8.14\ n-1\ times \\
rotate(s^n(0), x_{s^{n-1}(0)} :: \ldots :: x_0 :: (nil <> a)) & = & a <> \mathcal{L}(n) & \\
 & & & 8.13\ once \\
rotate(s^n(0), x_{s^{n-1}(0)} :: \ldots :: x_0 :: a) & = & a <> \mathcal{L}(n) & \\
 & & & 8.16 \\
rotate(s^{n-1}(0), x_{s^{n-2}(0)} :: \ldots :: x_0 :: a) <> (x_{s^{n-1}(0)} :: nil)) & = & a <> \mathcal{L}(n) & \\
 & & & 8.14\ n-2\ times \\
rotate(s^{n-1}(0), x_{s^{n-2}(0)} :: \ldots x_0 :: (a <> (x_{s^{n-1}(0)} :: nil))) & = & a <> \mathcal{L}(n) & \\
r(s^{n-2}(0), (x_{s^{n-3}(0)} :: \ldots :: x_0 :: (a <> (x_{s^{n-1}(0)} :: nil))) <> (x_{s^{n-2}(0)} :: nil)) & = & a <> \mathcal{L}(n) & \\
rotate(0, ((a <> (x_{s^{n-1}(0)} :: nil)) <> \ldots) <> (x_0 :: nil) & = & a <> \mathcal{L}(n) & \\
(\ldots (a <> (x_{s^{n-1}(0)} :: nil)) <> (x_0 :: nil) & = & a <> \mathcal{L}(n) &
\end{array}
$$

The rules used are:

$$
\underbrace{8.14\ \text{n-i times} + 8.13\ \text{once} + 8.16}_{repeat} + \overbrace{8.15 + \underbrace{8.14 + 8.13}_{repeat\ n-1}}^{can\ swap}
$$

or else:

$$
\underbrace{8.14 + 8.16 + 8.14 + 8.16 + \ldots}_{repeat} + \overbrace{8.15 + \underbrace{8.14 + 8.13}_{repeat\ n-1}}^{can\ swap}
$$

Note the correspondence between these alternative rule application sequences. They consist of essentially the same rules, but used in a different order. The main difference lies in the fact that in the case of the general proof of the generalised goal, the *rotate* definition (as given, but note that there are other forms) cannot be used immediately, and instead an *append* rewrite rule must be used to put the goal into the form to which the *rotate* definition could be applied. Now, if one could work out the nature of the transformation from the rules in the general proof of the original equation to those in the general proof of the generalisation, then it would be possible to use the EBG method on the rules in the latter to suggest a cut formula. (Or perhaps one should alter the goal before applying EBG, or modify the way EBG is carried out, for example by evaluating extra terms before applying a rule). The questions of when this new method should be used, how the inversion of the rules is relevant and whether the goal is a more general expression where the rules may be inverted will be considered below.

Now, the generalisation proves the original theorem by setting $a = nil$. What will the rules of the generalised general proof be in this case?

$$
rotate(s^n(0), (x_{s^{n-1}(0)} :: \ldots :: x_0 :: nil) <> nil) = nil <> \mathcal{L}(n) \quad (*)
$$

A new rule,

$$X <> nil \Rightarrow X \tag{8.19}$$

may be used, together with rule 8.13 to reduce $(*)$ to the original goal. Alternatively, the rules of the generalised general proof will carry through as normal. But 8.19 + 8.13 + the rules of the original general proof must be equivalent to the latter. So is there any way of getting from the original general proof to the generalised general proof? This is the proof transformation process of linearisation which will be considered with reference to examples in further sections.

Now, previously *lines* from the general proof have been inspected, and it has been attempted to linearise these in order to see the inductive process. Here, the *rules* of the general proof are going to be considered and altered, and EBG used on them to produce a cut formula. It has been shown above that the EBG in the case of accumulator examples, on the original general proof rules, will just return the original formula. Either this could be used, or else the linearisation methods detailed in Section 8.3 (by which $P(n)$ reduces to $P'(n-1)$, where P and P' have completely different structure), in order to suggest that the new methods detailed below should be used.

Alternatively, the structure of P' could itself suggest the reordering of the original rules in the general proof.

Let us recall the rules of the two general proofs:

general proof:

$$\underbrace{8.16 + 8.14}_{repeat} + \underbrace{8.14 + 8.13}_{repeat} + 8.15 \quad (A)$$

$$\text{or} \quad \underbrace{8.16 + \underbrace{8.14}_{repeat} + 8.13}_{} + 8.15 \quad (B)$$

general proof of cut formula:

$$\underbrace{\underbrace{8.14}_{repeat} + 8.13 + 8.16 +}_{repeat} \overbrace{\underbrace{8.14 + 8.13}_{repeat}}^{tidying\ up\ of\ expressions} + 8.15 \quad (C)$$

$$\text{or} \quad \underbrace{8.14 + 8.16}_{repeat} + \underbrace{8.14 + 8.13}_{repeat} + 8.15 \quad (D)$$

The similarity between the pairs (A) and (D), plus (B) and (C), is apparent.

So, by the methods mentioned above, if it has been ascertained that the cut formula must be of a different structure than the original formula, one could try a reordering of the rules of the general proof and then try EBG methods on this rule set to produce a cut formula.

Note that it is not a question of the permutation of rules being equivalent to the previous rule set (in the sense of resulting in the same final conclusions of rule blocks), for then one could have derived the cut formula directly from the general proof by EBG.

In this case, a permutation of the rules of the general proof should be tried, the result inspected, and then a proof given that the original formula is derivable from this. Now, there is a choice between doing an EBG from $X = Y$, or by starting with $rotate(len(f'(\mathcal{L}(n), a)), f(\mathcal{L}(n), a)) = f''(\mathcal{L}(n), a)$, and then working out what the functions f, f' and f'' might be.[7]

So, taking (A) with the first two rules swapped (or else just applying whatever rules are possible to $(*)$, in order), by the EBG method:

$$rotate(len(f'(\mathcal{L}(n), a)), f(\mathcal{L}(n), a)) = f''(\mathcal{L}(n), a)$$

[8.14 sets f to $\mathcal{L}(n) <> a$]

$$rotate(len(f'(\mathcal{L}(n), a)), x_{s^{n-1}(0)} :: (\mathcal{L}(n-1) <> a)) = f''(\mathcal{L}(n), a)$$

[use 8.16]

$$rotate(len(s^{-1}(f'(\mathcal{L}(n), a)), (\mathcal{L}(n-1) <> a) <> (x_{s^{n-1}(0)} :: nil)) = f''(\mathcal{L}(n), a)$$

$$\vdots \; repeat$$

$$rotate(len(s^{-n}(f'(\mathcal{L}(n), a)), ((\mathcal{L}(0) <> a) <> (x_{s^{n-1}(0)} :: nil) <> \ldots <> x_{s(0)} :: nil)) = f''(\mathcal{L}(n), a)$$

[8.14+ 8.13 — repeat]

$$rotate(\underbrace{len(s^{-n}(f'(\mathcal{L}(n), a)))}_{\equiv 0}, a <> (x_{s^{n-1}(0)} :: \ldots :: x_{s(0)} :: nil)) = f''(\mathcal{L}(n), a)$$

---

[7] Although in most cases these functions will be second order, they might also be higher-order; this would potentially cause problems in that unification might involve infinite digressions in some cases.

[use 8.15]

$$a <> (x_{s^{n-1}(0)} :: \ldots :: x_{s(0)} :: nil) = f''(\mathcal{L}(n), a)$$

$f'(\mathcal{L}(0), a) = nil$; set $f'(\mathcal{L}(n), a) = \mathcal{L}(n)$.

EQUALITY

[set $f''(\mathcal{L}(n), a) = a <> \mathcal{L}(n)$.]

It is then necessary to prove that this entails the original expression. If a is nil, then the original equation holds, but one must check the values for f, f' and f'' — in this case $X <> nil \equiv X$ (cf. equation 8.19) is also needed. Note that it will not be known if this will work out in advance, but there are usually only a small number of possibilities.

Now, it is known that the rules of the general proof are an injection into 'the rules of the general proof of the cut formula, together with the justification that the cut formula entails the original formula'. Hence, the rules of the general proof of the cut formula must be very similar to those of the general proof. When reordering is allowed, if this is carried out within the repetition blocks, this will be most likely to succeed. From other sources, it is known that a more general form of the object must be tried. So, one could try (proper) EBG with a rotation of the rules, and see if a suitable cut formula were reached (or, equivalently, using accumulators on the original expression). So really, the general procedure is to stick as close as possible to the general proof, but swap in the first rule block to obtain different structure in the original expression.

Hence, if using accumulators, there are two approaches. The first is a heuristic to try to construct a new general proof by means of permuting the rewrite rules in the original general proof. These rules would be applied either to $X = Y$ (cf. EBG method), or to a generalised original function (for example $rotate(len(f'(\mathcal{L}(n), a)), f(\mathcal{L}(n), a)) = f''(\mathcal{L}(n), a))$, and then the explanation-based generalisation method used to find f, f' and f''. Extra axioms such as $X <> nil = X$ are needed. The second approach is to inspect the original general proof and try to work out the cut formula from this directly. The first heuristic is worth a try, but I shall use the second approach, which is more rigorous.

In conclusion, I shall turn to the solution of trying to linearise the proof, by looking for repeated structure while allowing some generalisation.

8.4.3 Generalisation of Constants to Variables:

$$\forall l.\ rev2(l, nil) = rev(l)$$

Note that this example is also considered in Appendix C.3, with reference to showing that induction is blocked for the original formula but not for the suggested cut formula. The following rewrite rules are needed in addition to those given above:

$$rev(nil) \Rightarrow nil \tag{8.20}$$

$$rev(H :: T) \Rightarrow rev(T) <> (H :: nil) \tag{8.21}$$

$$rev2(nil, L) \Rightarrow L \tag{8.22}$$

$$rev2(H :: T, L) \Rightarrow rev2(T, H :: L) \tag{8.23}$$

The general proof is as follows:

$$\begin{aligned}
rev2(\mathcal{L}(n), nil) &= rev(\mathcal{L}(n)) \\
rev2(x_{s^{n-1}(0)} :: \mathcal{L}(n-1), nil) &= rev(x_{s^{n-1}(0)} :: \mathcal{L}(n-1)) \\
rev2(\mathcal{L}(n-1), x_{s^{n-1}(0)} :: nil) &= rev(\mathcal{L}(n-1)) <> (x_{s^{n-1}(0)} :: nil) \\
rev2(x_{s^{n-2}(0)} :: \mathcal{L}(n-2), x_{s^{n-1}(0)} :: nil) &= rev(x_{s^{n-2}(0)} :: \mathcal{L}(n-2)) <> (x_{s^{n-1}(0)} :: nil) \\
rev2(\mathcal{L}(n-2), x_{s^{n-2}(0)} :: x_{s^{n-1}(0)} :: nil) &= (rev(\mathcal{L}(n-2)) <> (x_{s^{n-2}(0)} :: nil)) <> (x_{s^{n-1}(0)} :: nil) \\
&\ \ \vdots \quad \textit{similarly, } n-2 \textit{ times} \\
rev2(\mathcal{L}(0), x_0 :: \ldots :: x_{s^{n-1}(0)} :: nil) &= (rev(\mathcal{L}(0)) <> x_0 :: nil) <> \ldots <> (x_{s^{n-1}(0)} :: nil) \\
rev2(nil, x_0 :: \ldots :: x_{s^{n-1}(0)} :: nil) &= (\ldots (nil <> x_0 :: nil) <> \ldots) <> (x_{s^{n-1}(0)} :: nil) \\
x_0 :: \ldots :: x_{s^{n-1}(0)} :: nil &= x_0 :: \ldots :: x_{s^{n-1}(0)} :: nil
\end{aligned}$$

The following holds:

$R(n) \equiv rev2(\mathcal{L}(n), nil) = rev(\mathcal{L}(n))$

$R'(n-1) \equiv rev2(\mathcal{L}(n-1), x_{s^{n-1}(0)} :: nil) = rev(\mathcal{L}(n-1)) <> (x_{s^{n-1}(0)} :: nil)$

The generalisation suggested is

$$rev2(l, a) = rev(l) <> a.$$

This is the correct one. Note that this method has enabled the generalisation of *nil* (a constant) to a variable. Note also that the same block of rules is being

carried out a set number of times, such as n, which is analogous to the same rule being applied n times in the general proofs of natural number examples.

Hence, the heuristic for this case is:
"to provide the cut formula, retain structure common to R and R' the same, do induction on l if $\mathcal{L}(s(n))$ is at the same structural position in $R(s(n))$ as $\mathcal{L}(n)$ is in $R'(n)$, and generalise to a new variable any other common subexpressions in R''"

In a further section these heuristics will be conjoined, and a more general method will be seen to emerge.

Generalising from $Q(0)$

If the general proof is thought of as being linear in some way, it might also be interesting to consider $Q(0)$ (ie. $R'^{\cdots'}(0)$), because it is hard to describe $Q(m)$ in terms of $Q(m-1)$. For this example,

$$Q(0) \equiv rev2(\mathcal{L}(0), x_0 :: \ldots :: x_{s^{n-1}(0)} :: nil)$$

$$= (rev(\mathcal{L}(0)) <> x_0 :: nil) <> \ldots <> (x_{s^{n-1}(0)} :: nil)$$

Note that it is not desirable to replace $\mathcal{L}(0)$ by nil here (although this may be necessary in order to justify the final equation), since the idea is to generalise from $Q(0)$. In order to generalise, one would wish to rearrange expressions so that similar terms were on both sides of the equation. It is necessary to prove some extra information (which could be stored as it reoccurs again and again). Using $(H :: T) <> X \Rightarrow H :: (T <> X)$ and $(X <> Y) <> Z \Rightarrow X <> (Y <> Z)$ (which is provable by induction), then:

$(X <> (H :: nil)) <> H2 :: nil = X <> ((H :: nil) <> H2 :: nil)$
$= X <> (H :: (nil <> H2 :: nil))$
$= X <> (H :: H2 :: nil)$
Continuing in this manner, there is on the right hand side,

$$(rev(\mathcal{L}(0))) <> \underbrace{x_0 :: \ldots x_{s^{n-1}(0)} :: nil}_{a}$$

Thus, $Q(0) \equiv rev2(\mathcal{L}(0), a) = rev(\mathcal{L}(0)) <> a$. From this, the generalisation $rev2(l, a) = rev(l) <> a$ is suggested. Note however that it was necessary to prove various lemmata in order to be able to do this (cf. Section 8.6, which discusses control of the application of lemmata).

8.4.4 Generalisation of Terms to Variables:

$$\forall l.\ rev(rev(l) <> y :: nil) = y :: rev(rev(l))$$

The general proof runs as follows:

$$\begin{aligned}
rev(rev(\mathcal{L}(n)) <> y :: nil) &= \\
y :: rev(rev(\mathcal{L}(n))) & \\
rev(rev(x_{s^{n-1}(0)} :: \mathcal{L}(n-1)) <> y :: nil) &= \\
y :: rev(rev(x_{s^{n-1}(0)} :: \mathcal{L}(n-1))) & \\
rev((rev(\mathcal{L}(n-1)) <> x_{s^{n-1}(0)} :: nil) <> y :: nil) &= \\
y :: rev(rev(\mathcal{L}(n-1)) <> x_{s^{n-1}(0)} :: nil) & \\
\vdots \quad similarly & \\
rev(((x_0 :: nil) <> \ldots <> (x_{s^{n-1}(0)} :: nil)) <> (y :: nil)) &= \\
y :: rev((x_0 :: nil) <> \ldots (x_{s^{n-1}(0)} :: nil)) & \\
\vdots \quad by\ 8.20,\ 8.21\ \&\ 8.14 & \\
y :: x_{s^{n-1}(0)} :: \ldots :: x_0 &= \\
y :: x_{s^{n-1}(0)} :: \ldots :: x_0 &
\end{aligned}$$

In this example, it is the case that:

$R(n) \equiv rev(rev(\mathcal{L}(n)) <> y :: nil) = y :: rev(rev(\mathcal{L}(n)))$

$R'(n-1) \equiv rev((rev(\mathcal{L}(n-1)) <> x_{s^{n-1}(0)} :: nil) <> y :: nil)$

$= y :: rev(rev(\mathcal{L}(n-1)) <> x_{s^{n-1}(0)} :: nil)$ The procedure is to leave structure common to R and R' the same, do induction on l if $\mathcal{L}(n)$ is at the same structural point in $R(n)$ as $\mathcal{L}(n-1)$ in $R'(n-1)$, and to generalise any other common subexpressions in R'.

This suggests the generalisation

$$rev(a <> y :: nil) = y :: rev(a)$$

This is the correct generalisation, but note that other methods could have reached it more quickly in this case by generalisation of common subexpressions initially (cf. (Boyer & Moore, 1979), where this example is considered). Thus, this example does not fail by means of contravening the restriction placed in Subsection 8.1.2

that redundant rules in the general proof should be omitted, but rather provides a demonstration that, although the method works, it is in this case not the best generalisation method to use. However, note that the structure of the proof has actually been investigated and it has been checked that such a generalisation (from a linear general proof) is the correct one and so will not produce a non-theorem.

Generalising from $Q(0)$

$Q(0) \equiv rev((rev(\mathcal{L}(0)) <> x_0 :: nil) <> \ldots <> (x_{s^{n-1}(0)} :: nil)) <> y :: nil)$
$= y :: rev((rev(\mathcal{L}(0)) <> (x_0 :: nil)) <> \ldots <> x_{s^{n-1}(0)} :: nil))$
which reduces to:
$rev(((rev(\mathcal{L}(0)) <> (x_0 :: \ldots :: x_{s^{n-1}(0)} :: nil)) <> y :: nil)$
$= y :: rev(rev(\mathcal{L}(0)) <> (x_0 :: \ldots :: x_{s^{n-1}(0)} :: nil))$
Now, in fact the suggested generalisation above should have initially been of

$$rev((rev(l)) <> a) <> y :: nil) = y :: (rev(rev(l) <> a))$$

This is what is suggested in this case, too. b is set to $rev(l) <> a$, which suggests

$$rev(b <> y :: nil) = y :: rev(b)$$

However, if substitutions of this sort are going to be carried out, one might as well have put $b = rev(l)$ in the original equation! There is a gain, though, in the sense that by inspection of the general proof, it will be known that this generalisation will work.

8.4.5 Summary

To summarise, a general representation for lists has been defined, and various methods suggested for the provision of a cut formula via the general proof. Note that in fact a cut formula for the examples in Subsection 8.1.5 could also be provided by the method described in this section, which subsumes the original suggestion. However, there are problems with the comparison of $R(n)$ with $R'(n-1)$. It may be argued that one should instead compare $R(n)$ with $R'^{\cdots'}(0)$, since there

may be special rules simplifying expressions at the end of the general proof. However, it should be possible to get round this problem if it is insisted that similar constants on each side of the equality within $R'(n-1)$ would be identified this way via using appropriate rewrite rules. Using the method described above, in many differing types of cases a correct cut formula may be found.

The generalisation of $R'^{\cdots\prime}(0)$ corresponds to linearising the structure of the general proof as follows. $R'^{\cdots\prime}(0)$ is inspected, l is substituted for $\mathcal{L}(0)$, and any new expressions introduced (compared with $R(n)$) are rewritten to the most general form, with equivalent expressions either side of the equality. This captures the whole transformation occurring in the general proof. This process is in fact reached in stages, with equal blocks of rules used for each stage, and it is the representation for this (linear) process which I hope to capture. In other words, there is a sort of induction going on, as may be seen from the equal blocks of rules, but this is not directly seen in the original general proof. However, the generalisation of $R'^{\cdots\prime}(0)$ is exactly that which captures this induction; by the way it has been formed, induction on it will be the latent induction of the general proof, and so it may be proved by just one induction. This corresponds to a linearisation of the structure of the general proofs.

8.5 Summary of Linearisation Approach

This section provides a summary of how the transformation of the general proof into a linear form is made. As discussed in Section 8.3, if the proof is linear induction may be carried out directly. If the proof is not linear there are two cases:

- The explanation-based generalisation method may be used to generate a linear general proof.
- Otherwise, linearisation of the proof is attempted by looking for repeated structure while allowing some generalising.

(Note that when making the transformation, it might become apparent that a new lemma would have to be cut into the proof in PA, most commonly an associativity or commutativity axiom. The axioms needed in the proof in PA will also be needed to obtain equality in the general proof. Section 8.6 discusses this in further detail.) The following are further examples of various types of proofs, and illustrate how the linearisation process may be carried out.

8.5.1 Easily Linearisable Proofs

- $\forall x\ (x+x)+x = x+(x+x)$

  It is necessary to define $Q(n)$ and the rule block i (cf. discussion of this example in Subsection 8.3.3).

  $$Q(x,y,z) = \forall x\ \forall y\ \forall z\ (x+y)+z = x+(y+z)$$

  (So $Q(x,x,x) = \forall x\ (x+x)+x = x+(x+x)$.)

  An alternative arrangement of rules for this proof is:

$$\begin{array}{rcrcl}
Q(n,n,n) & \equiv & (s^n(0)+s^n(0))+s^n(0) & = & s^n(0)+(s^n(0)+s^n(0)) \\
\text{rule} & & s(s^{n-1}(0)+s^n(0))+s^n(0) & = & s(s^{n-1}(0)+(s^n(0)+s^n(0))) \\
\text{block i} & & s((s^{n-1}(0)+s^n(0))+s^n(0)) & = & s(s^{n-1}(0)+(s^n(0)+s^n(0))) \\
Q(n-1,n,n) & \equiv & (s^{n-1}(0)+s^n(0))+s^n(0) & = & s^{n-1}(0)+(s^n(0)+s^n(0)) \\
\text{repeating i} & & & \vdots & \\
Q(0,n,n) & \equiv & (0+s^n(0))+s^n(0) & = & 0+(s^n(0)+s^n(0)) \\
\text{rules j} & & & \vdots & \\
 & & s^n(0)+s^n(0) & = & s^n(0)+s^n(0)
\end{array}$$

  I shall consider the questions of what permutation of the original rules to this rule block is allowed, and also how far the linearisation process may be automated. Note that cancel represents the use of the cancellation rule $s(x) = s(y) \Rightarrow x = y$.

  Original Rules:

  $6.2(n)$, $6.1(1)$, $6.2(n)$, $6.1(1)$, $6.2(n)$, $\mathsf{cancel}(n)$, $\mathsf{cancel}(n)$

  Rule Block i:

  $6.2(1)$, $6.2(1)$, $6.2(1)$, $\mathsf{cancel}(1)$, $\mathsf{cancel}(1)$

  Rule Block j:

6.1(1), 6.1(1)

(Note that the rules are applied in the same relative subpositions in both general proofs, but I have not bothered to distinguish the subterms with this representation).

The generalisation may be detected from this new (linearised) proof. Alternatively, the EBG method could be used, using the previous set of rules. Perhaps the way of trying to linearise the proof is, given rules:

$$A(n),\ B,\ C(n),\ D,\ E,\ F(n),\ \textit{etc.}$$

to apply A once, try to apply B, else C, then try to apply D, etc. The rule blocks would be expected to be:

$\underline{i}$: A, C, F.

$\underline{j}$: B, D, E.

Linearisation here has a symbolic appearance, namely to group together the rules which are applied repeatedly, and those which are only applied once. If this gives a complete general proof, all variables are generalised apart, the rewrite rules i are performed, and then the result compared with the original.

eg. $Q(n', m, p, q, r, t) \equiv (s^{n'}(0) + s^m(0)) + s^p(0) = s^q(0) + (s^r(0) + s^t(0))$

converts by rules i to:

$Q(n'-1, m, p, q-1, r, t) \equiv (s^{n'-1}(0)+s^m(0))+s^p(0) = s^{q-1}(0)+(s^r(0)+s^t(0))$

Also, $Q(\vec{0}) \equiv (s^{n'-n}(0) + s^m(0)) + s^p(0) = s^{q-n}(0) + (s^r(0) + s^t(0))$,

which gives equality using 6.1 by setting $n' = n$, $q = n$, $m = r$, $p = t$.

So, $Q(x) = \forall x\ (x + y) + z = x + (y + z)$ is the generalisation.

- $\underline{\forall x\ x.(x + x) = x.x + x.x}$

  General Proof:

$$\begin{array}{rcll}
\underline{s^n(0).(s^n(0) + s^n(0))} & = & s^n(0).s^n(0) + s^n(0).s^n(0) & 8.10(n) \\
\underline{0.(s^n(0) + s^n(0))} + \underbrace{(s^n(0) + s^n(0))}_{n} & = & \ldots & 8.9; 6.1 \\
\underbrace{(s^n(0) + s^n(0))}_{n} & = & \underline{s^n(0).s^n(0)} + \underline{s^n(0).s^n(0)} & 8.10(n); \\
\ldots & = & (\underline{0.s^n(0)} + \underbrace{s^n(0)}_{n}) + (\underline{0.s^n(0)} + \underbrace{s^n(0)}_{n}) & 8.9; 6.1 \\
\ldots & = & \underbrace{s^n(0)}_{n} + \underbrace{s^n(0)}_{n} & \\
\multicolumn{4}{l}{\textit{EQUALITY}}
\end{array}$$

The equality is achieved in the last line using the associativity of plus, $A + (B+C) = (A+B)+C$, in order to show $(s^n(0)+s^n(0))+\ldots+(s^n(0)+s^n(0)) = (((s^n(0)+s^n(0))+s^n(0))+\ldots)+(((s^n(0)+s^n(0))+s^n(0))+\ldots)$

An alternative arrangement of rules is:

$s^n(0).(s^n(0)+s^n(0)) = s^n(0).s^n(0)+s^n(0).s^n(0)$

$s^{n-1}(0).(s^n(0)+s^n(0))+(s^n(0)+s^n(0))\ldots$ *by* 8.10(1)

$\ldots = (s^{n-1}(0).s^n(0)+s^n(0))+(s^{n-1}(0).s^n(0)+s^n(0))$ *by* 8.10(1); 8.10(1)

$\ldots = (s^{n-1}(0).s^n(0)+s^{n-1}(0).s^n(0))+(s^n(0)+s^n(0))$ using ass-of-plus and comm-plus (or alternatively, $(A+B)+(C+D) = (A+C)+(B+D)$)

ie. $s^{n-1}(0).(s^n(0)+s^n(0)) = s^{n-1}(0).s^n(0)+s^{n-1}(0).s^n(0)$ using $A+B = C+B \Rightarrow A = C$.

So, $Q(x) \equiv x.(y+z) = x.y+x.z$ reduces to $Q(x-1)$, given these extra rules (ie. commutativity).

Also, $Q(0) \equiv 0.(s^n(0)+s^n(0)) = 0.s^n(0)+0.s^n(0)$, which reduces to $0 = 0$ using 8.9 and 6.1.

Original Rules:

8.10(n), 8.9(1), 6.1(1), 8.10(n), 8.10(n), 8.9(1), 8.9(1), 6.1(1), 6.1(1), ass-of-plus(n)

Rule Block i:

8.10(1), 8.10(1), 8.10(1), ass-of-plus, comm-plus $*$

Rule Block j:

8.9(1), 6.1(1), 8.9(1), 8.9(1), 6.1(1)

$*$ shows that associativity of plus and commutativity of plus are needed in the inductive proof, to get the final answer. This is suggested by the fact that associativity of plus is used in the original proof, which shows that there will have to be some forms of rewriting at the end.

Note that the linearisation follows the pattern suggested in the previous subsection.

- $\forall l\ (l <> l) <> l = l <> (l <> l)$

Note that the definitions used in the arithmetical examples are slightly different than those used for lists: in the latter, the structure is released slowly, rather than all at once. That is to say, in the former $\underline{n}$ is replaced by $s^n(0)$ and not $s.\underline{n-1}$, which is the equivalent of what has been substituted in the list examples. This is primarily for brevity, since it is easier to write $x_{s^{n-1}(0)} :: \mathcal{L}(n-1)$ than $x_{s^{n-1}(0)} :: \ldots :: x_0 :: nil$. In addition, it will alter the structure of the general proof — for lists, the linear structure will become much more apparent.

With the example above, the general proof corresponds to the alternative way of writing the blocks of repeated rules i, and then finally the rules j to reach equality. Thus, the generalisation of $Q(n) \equiv (\mathcal{L}(n) <> \mathcal{L}(m)) <> \mathcal{L}(m) = \mathcal{L}(n) <> (\mathcal{L}(m) <> \mathcal{L}(m))$ may be read directly from the general proof. From the fact that $Q(r)$ reduces to $Q(r-1)$ in the general proof for $r \leq n$, and also that $Q(0)$ is provable, the general proof is in a linear form, and also Q will be a suitable candidate for induction.

Alternatively, Q can be obtained by EBG, and the resulting general proof checked to see that it is in a linear form (ie. that that generalisation will be amenable to induction).

8.5.2 More Difficult Examples

These are cases when EBG on the original rule set will not suggest an appropriate induction. Hence, the process of linearisation of the general proof is particularly appropriate in this case as a method of suggesting a cut formula.

Such a process is illustrated by the $len(rev(l)) = len(l)$ example:

by rule block i

$len(rev(\mathcal{L}(n))) = len(\mathcal{L}(n))$ reduces to

$len(rev(\mathcal{L}(n-1)) <> x_{s^{n-1}(0)}) = len(x_{s^{n-1}(0)} :: \mathcal{L}(n-1))$

by rule block j

$len((rev(\mathcal{L}(0)) <> (x_0 :: nil)) <> \ldots <> x_{s^{n-1}(0)} :: nil) = len(x_{s^{n-1}(0)} :: nil)$

$len((nil <> (x_0 :: nil)) <> \ldots) = \ldots$

$len((x_0 :: nil <> x_{s(0)} :: nil) <> \ldots <> x_{s^{n-1}(0)} :: nil) = \ldots$

$len(x_0 :: (nil <> x_{s(0)} :: nil) <> \ldots) = \ldots$

$len(x_0 :: x_{s(0)} :: \ldots) = len(x_{s^{n-1}(0)} :: \ldots :: nil)$

$\underbrace{s \ldots s}_{n} len(nil) = \underbrace{s \ldots s}_{n} len(nil)$

$s^n(0) = s^n(0)$

EQUALITY

Note that

$Q(r) = len((rev(\mathcal{L}(r) <> x_{s^r(0)} :: nil)) <> \ldots <> x_{s^{n-1}(0)} :: nil)$

$= len(x_{s^{n-1}(0)} :: \ldots :: x_{s^r(0)} :: \mathcal{L}(r))$

The current goal is to gather together the x_i into a new list a. $Q(r)$ will be the generalisation, since the proof described by Q is linear. The following transformation is being performed on the general proofs:

$$\begin{array}{ccc} P(x,\vec{x}) & & P(x,\vec{y}) \\ P(x-1,\vec{x}) & \Rightarrow & P(x-1,\vec{y}) \\ \vdots & & \vdots \\ P(0,\vec{x}) & & P(0,\vec{y}) \end{array}$$

where $P(x,\vec{x})$ is for example $P(x,x,x)$ and $P(x,\vec{y})$ is for example $P(x,y,z)$. The structure of P is not apparent from the first line of the proof, since for $r = n$, the extra structure will not yet appear.

Q is a different structure to P, and hence the EBG method will not work.

General Proof:

$$\dfrac{\dfrac{\dfrac{Q(n)\ \text{(original goal)}}{Q(r)}\,i}{Q(r-1)}\,i}{\begin{array}{c}\vdots\, i \\ Q(0)\end{array}}\,j$$
$$EQUALITY$$

suggests the general proof below, which is linearisable, and therefore the associated formula is suitable as an induction candidate.

General Proof:

Theorem	Cut Formula
$\forall x\ (x+x)+x = x+(x+x)$	$\forall x \forall y \forall z\ (x+y)+z = x+(y+z)$
$\forall x\ x+s(x) = s(x+x)$	$\forall x \forall y\ x+s(y) = s(x+y)$
$\forall x\ x+s(x) = s(x)+x$	$\forall x \forall y\ x+s(y) = s(x)+y$
$\forall x\ x.(x+x) = x.x+x.x$	$\forall x \forall y \forall z\ x.(y+z) = x.y+x.z$
$\forall x\ (x+x).x = x.x+x.x$	$\forall x \forall y \forall z\ (x+y).z = y.z+x.z$
$\forall x\ (2+x)+x = 2+(x+x)$	$\forall x \forall y\ (2+x)+y = 2+(x+y)$
$\forall x\ \forall y\ (x+y)+x = x+(y+x)$	$\forall x \forall y \forall z\ (x+y)+z = x+(y+z)$
$\forall x\ \neq 0 \rightarrow p(x)+s(s(x)) = s(x)+x$	$\forall x \forall y\ x \neq 0 \rightarrow p(x)+s(s(y)) = s(x)+y$
$\forall x\ even(x+x)$	$\forall x\ even(2.x)$
$\forall l\ len(rev(l)) = len(l)$	$\forall l\ len(rev(l) <> a) = len(rev(a) <> l)$
$\forall l\ rotate(len(l), l) = l$	$\forall l\ rotate(len(l), l <> a) = a <> l$
$\forall l\ rev(rev(l) <> y :: nil) = y :: rev(rev(l))$	$\forall l\ rev(a <> y :: nil) = y :: nil$
$\forall l\ rev2(l, nil) = rev(l)$	$\forall l\ rev2(l, a) = rev(l) <> a$
$\forall l\ (l <> l) <> l = l <> (l <> l)$	$\forall l \forall p \forall q\ (l <> p) <> q = l <> (p <> q)$

Table 8–2: Cut Formulae Suggested by Guiding Method for Various Examples

$$\dfrac{\begin{array}{c} Q(r) \\ \vdots \\ Q(0) \end{array}}{EQUALITY}\ j$$

To get back from $Q(\underline{r})$ to a generalisation not dependent on those particular x_i, it is necessary to clump together all the x_i into a list a. Since $\forall x_i$ holds in the first place, and it is true for all n, then the generalisation will be true for all a. Even if it is not, one could generalise $\forall x\ \exists a\ Q(x,a)$ to $\forall x\ \forall a\ Q(x,a)$, and use the latter as the cut formula.

8.5.3 Suggested Cut Formulae

The revised solutions using this method for the various list examples considered in this chapter are:

- $Q(r) \equiv len(((rev(\mathcal{L}(r))) <> x_{s^r(0)} :: nil) <> \ldots <> x_{s^{n-1}(0)} :: nil) = len(x_{s^{n-1}(0)} :: \ldots :: x_{s^r(0)} :: \mathcal{L}(r))$
  suggests a (correct) generalisation of $len(rev(l) <> a) = len(rev(a) <> l)$, for which step it is necessary to use the associativity of append (for the left-hand side).

- $Q(r) \equiv rotate(len(\mathcal{L}(r)), (((\mathcal{L}(r) <> x_{s^n(0)} :: nil) <> \ldots <> (x_{s^r(0)} :: nil)))) = x_{s^n(0)} :: \ldots :: x_{s^r(0)} :: \mathcal{L}(r)$ suggests a (correct) generalisation of $rotate(len(l), l <> a) = a <> l$. Again, the associativity of append must be used.

- $Q(r) \equiv rev((rev(\mathcal{L}(r)) <> (x_{s^r(0)} :: nil) <> \ldots <> (x_{s^{n-1}(0)} :: nil) <> (y :: nil))) = y :: rev(\ldots(rev(\mathcal{L}(r)) <> (x_{s^r(0)} :: nil)) <> \ldots) <> (x_{s^{n-1}(0)} :: nil))$ suggests a generalisation of $rev(rev(l) <> a) <> y :: nil) = y :: rev(rev(l) <> a)$ (using associativity of append). This is if just the a is replaced. But in fact it is desirable to replace as much common structure as possible, since $\mathcal{L}(r)$ is only a random list anyway, and so instead $rev(l) <> a$ (which occurs on both sides) should be replaced to get $rev(b <> y :: nil) = y :: rev(b)$, which is the best generalisation. It would have been possible to have reached this generalisation from the original formula by replacing common subterms by a new variable, but this heuristic can result in overgeneralisation, and at least in this case it is known that this result is a correct generalisation in the sense that induction upon it will carry through, so that is an improvement.

- $Q(r) \equiv rev2(\mathcal{L}(r), x_{s^r(0)} :: \ldots :: x_{s^{n-1}(0)} :: nil) = (rev(\mathcal{L}(r)) <> (x_{s^r(0)} :: nil)) <> \ldots <> (x_{s^{n-1}(0)} :: nil)$ suggests a (correct) generalisation of $rev2(l, a) = (rev(l)) <> a$.

In each case $x_{s^r(0)} :: \ldots :: x_{s^{n-1}(0)} :: nil$ has been replaced by a. Recall that $\mathcal{L}(r)$ is in fact by definition $x_0 :: \ldots x_{s^{r-1}(0)} :: nil$. The final stages of generalisation, with rearranging to achieve equality, were achieved by guessing (since there are only a small number of possibilities, and pre-stored information could be used, since the same or similar justifications arise each time), and then checking this result. As

mentioned in Section 8.4.2, there are only a small number of possibilities, and these may be found, and checked, mechanically. Automation of linearisation is possible, since there is literal repetition of rules (with parameters denoting where and the number of times they are applied), and from these rule blocks may be formed, if necessary by rearranging the rules. The rth case provides the generalisation, by means of the collection and rearrangement of terms, as described above.

In conclusion, the approach also applies generally to many data-types. Not only is it the case that certain new structural patterns may be seen in the general proof which may guide generalisation, but also that the general representation of an arbitrary object of that type (eg. $s^n(0)$ for natural numbers, $[x_1, x_2, \ldots x_m]$ for lists, etc.) enables the structure of that particular data-type to be exploited, in the sense that rewrite rules may be used which would not otherwise be applicable. For the natural number examples given in this chapter, the general proof is linear in the sense that the proof of $P(s(n), n)$ reduces to that of $P(n, n)$. Renaming of variables via explanation-based generalisation results in a general proof of form $Q(n)$ reducing to $Q(n-1)$. However, in many examples involving lists, this is not so, and a new method for providing a cut formula is needed. The result is an approach which subsumes the previous suggestion: the linearisation of the general proof. The general proof is put into a form such that there is a repeated rule block i. Thus the rth line after r uses of i is $Q(r)$, such that $Q(r)$ reduces to $Q(r-1)$. $x_{s^r(0)} :: \ldots x_{s^{n-1}(0)} :: nil$ is replaced by a in $Q(r)$, and rearrangements made as appropriate, to suggest a generalisation for which induction will carry through. In this way, correct cut formulae are suggested for the list examples considered in this chapter. In particular, one generalisation provided by this method of $\forall a\ \forall l\ len(rev(l) <> a) = len(rev(a) <> l)$ (from $len(rev(l)) = len(l)$) is a better result (since it only requires one induction) than the only alternative suggestion provided for this example of $\forall a\ \forall l\ len(rev(l) <> a) = len(a <> l)$[8] (requiring two inductions).

---

[8]Provided by Jane Hesketh's method (Hesketh, 1991b); see Subsection 11.3.3.

In examples where none of the heuristics suggested in this chapter work, it may still be appropriate to examine the general proof to see if some new structure has emerged. For example, the general proof of $even(s^n(0) + s^n(0))$ reduces to that of $even(s^{2n}(0))$, which suggests a cut formula of $even(2.x)$, with induction on x (see Section 9.1). Such examples, where these heuristics fail and yet some information is provided by the general proof, are considered in Chapter 9.

8.6 Requirement for the Use of Lemmata

The question to be considered in this section is that of when lemmata such as the associativity of append or the commutativity of plus are needed in the (linearised) general proofs.

As seen from the previous sections, the associativity of append (cf. Axiom C.1 of Appendix C) is needed for all of the generalised list proofs. This is because there is a definitional position in all of the examples in which the appropriate rewrite rule cannot be used until the structure of the term in this position has been broken down, or else it is necessary to use the definition of append n times in the rule block j. If a rule is used n times in j, that means that the structure cannot be distributed earlier in the proof as normal, in other words into the rule block i (as considered above) one rule each time. So if this rule is used n times in j, it means that it is necessary to use an associativity rewrite rule whenever the generalisation stage is carried out. In other words, as seen from the examples considered in this chapter, if an additional lemma is used in a general proof, there will have to be some forms of rewriting at the end of the corresponding linearised general proof in which such lemmata will also be needed.

Note how this corresponds to the use of cancellation in arithmetical proofs. What is normally referred to as cancellation is the rule $s(X) = s(Y) \Rightarrow X = Y$ (or the equivalent rule for the operator under consideration). This rule is essentially redundant in the general proofs, since equality may be achieved without it (since rules may be applied at subpositions of formulae), and is normally used just to

condense lines of a general proof. A rule which is often useful in general proofs, and which may be necessary to achieve the final equality, is the associativity of times: $(A.B).C = A.(B.C)$, which compares directly with the associativity of append: $(A <> B) <> C = A <> (B <> C)$.

From the examples considered in this chapter, it is apparent that associativity brings benefits as far as theorem-proving is concerned: use of associativity of plus or append, as appropriate, enables linearisation of the general proof, and hence the suggestion of a cut formula.

8.6.1 The Orientation of Rewrite Rules

The rewrite rules are set to be in a pre-defined direction (to give a normalisation procedure and avoid looping). The orientation of recursive definitions are set in the direction $s(X) + Y \Rightarrow s(X + Y)$ and $0 + Y \Rightarrow Y$, etc., although it is possible for the user to change this if desired. Moreover, the same approach is adopted by Boyer and Moore, in that associativity of append is oriented in one direction only in each example (in practice this works, since for none of the examples considered is use of associativity in both directions needed in the same proof).

8.6.2 Circularity

The question might be raised about whether there is circularity in the generalisation method suggested in this chapter, in the sense that the same problem is being re-represented at the meta-level in the general proofs. Such circularity would be obviously present if an original goal of $x + (y + 1) = (x + 1) + y$ reduced finally to $s^{x+(y+1)}(0) = s^{(x+1)+y}(0)$ in the general proof. Although general proofs might be carried out in such an entirely circular manner, as for example, counting the number of brackets of a numeral representation of a number, this is not the case with the representation presented. For example, the general proof of $(x+x)+x = x+(x+x)$ reduces to $s^n(s^n(0)+s^n(0)) = s^n(s^n(0)+s^n(0))$, and not $(s^n(s^n))s^n(0) = s^n(s^n(s^n(0)))$. When charged with circularity in the proofs, for

instance when showing equality from $s^p.s^r(0) = s^r.s^p(0)$, it is often only the use of an associativity or commutativity rule as a rewrite rule which is occurring. So long as this is does not occur in the general proof of the corresponding associativity or commutativity rule, its use is not necessarily circular.

Sometimes within the general proofs some notation may be used as shorthand, and the impression given that something subtle is being carried out at the meta-level, for example $s^{x+y}(z)$ written instead of $s^x.s^y(z)$. However, this is just a notational convenience rather than a logical step. What is important is to understand the relevant syntactical definitions: in this case $s^n(x)$ is defined recursively as a shorthand for $\underbrace{s(\ldots s}_{n\ times}(x))\ldots)$ (cf. the definition of numeral_string: $nat \times string \rightarrow string$ in Section 5.2.1). Therefore, $s^x.s^y(z)$ will represent $\underbrace{s(\ldots s}_{x}(\underbrace{s(\ldots s}_{y}(z))\ldots)$. Thus **Note 2** of Subsection 8.1.3 does not introduce circularity, but merely makes use of such notation: it does not reformulate the same question, as may be shown by the fact that $x + s(x) = s(x) + x$ is not provable by induction, whereas $s(s^n(x)) = s^n(s(x))$ is provable by induction on n.

However, circularity might occur if a statement such as "if rule R is applied n times at Pos, the result is Seq" were proved via the use of the cut rule, and hence the associated generalisation problems. However it is not, and although we do not *know* that we have the property of non-circularity, standard examples such as those given in the Appendix and in Chapter 8 show that generalisation is not needed in order to form the general proof in normal practice, and provide empirical evidence that proofs *are* easier to obtain in $PA_{c\omega}$ than in PA. Analogously, the fact that there has been above an appeal to induction over trees to show correctness raises the question of whether there is a problem of circularity, but again, in practice such meta-induction applies to simpler problems, the generalisation problem does not arise and there is a gain after all.

To summarise, in the examples given in this chapter especially the question of circularity arises — is the general proof just reducing the problem to what we were trying to prove in the first place? This charge may be drawn against some methods of provision of a general proof, but in the main it is not true, since the

general proof can usually be completed using just the original rewrite rules given plus associativity or commutativity (cf. Sections 6.4.1 and 8.6).

8.7 Final Conclusions

A new method of generalisation has been proposed which has met with a substantial amount of success. Three main approaches have been described in this chapter. The first is a generalisation heuristic which consists in renaming unaltered variables in the general proof. This approach has been implemented, but was dropped as it was subsumed by the second method. The latter is a form of explanation-based generalisation, which works if the generalisation required is the most general form of the original goal. This approach has the advantage that the generalisation suggested is shown to be the most general form of the original goal (having that structure). This method has been implemented (see Appendix E), and produces more general generalisations than the first approach. A third, yet more general, approach (which has not yet been implemented) is that of linearising the general proof. If accumulators are needed, or in some complicated cases, the linearisation approach suggested is able to provide a suitable cut formula. Moreover, when a cut formula is suggested, it will have been shown via linearisation of the general proof that induction may be successfully carried out upon it. Hence, it is not necessary to actually carry out the induction to know that the proof may be completed by using the cut rule and induction.

Chapter 9

Extension of the Generalisation Method

"Generalisation is necessary to the advancement of knowledge."

Macaulay

This chapter extends the generalisation method suggested in the previous chapter to more speculative examples than the latter. Although there is a general approach, it does not work in all cases; even so, a pattern is seen to emerge, and this may be helpful to the user. In some cases it is not appropriate to look for generalisation but an ω-proof may still be provided.

9.1 Indication of a Cut Formula

A selection of the theorems proved automatically by the generalisation method suggested in the previous chapter is listed in Table 8–1. Note that these examples involve generalisation apart, generalisation of common subexpressions, or else introduction of accumulators. In these cases the method works fairly straightforwardly. Although the examples listed in the table are of a similar simple form, this method may also be applied to complicated examples containing nested quantifiers, etc., for the ω-rule applies to arbitrary sequents (see Section 9.6). For the example $\forall x\ even(x + x)$, which is considered below, a cut formula of $even(2.x)$

may be extracted from the form of the general proof, which is an improvement over other methods (see Chapter 11 for a discussion of this example). However, in some cases where an ω-proof may be provided, it is not clear what the cut formula might be. The following are some examples in which a general proof is (informally) suggested, together with cut formulae if appropriate. However, the algorithms of the previous chapter do not apply rigorously to these examples, and it is more a question of the general proof indicating a generalisation.

The following are examples which demonstrate how the general proof may provide information which other methods do not, even if the guiding method as described in the previous chapter does not carry through directly.

1. $\underline{\forall x\ x + x = 0 \rightarrow x = 0}$

   For this example the step case of induction is trivially true, since $s(n) \neq 0$ for any n. The general proof is given below (there is a casesplit: the case for $n = 0$ involves proving that $0 + 0 = 0$, and the case for $n \neq 0$ is as follows):

$$\begin{array}{rcll} \underline{n} + \underline{n} = 0 & \rightarrow & \underline{n} = 0 & \\ s^n(0) + s^n(0) = 0 & \rightarrow & s^n(0) = 0 & \\ s^n(s^n(0)) = 0 & \rightarrow & s^n(0) = 0 & 6.1,\ 6.2 \end{array}$$

   This "suggests" the generalisation $\forall x\ x + y = 0 \rightarrow x = 0$, which is the correct one.

2. $\underline{\forall x\ even(x + x)}$

   Given the axioms and general proof as follows:

   The following axioms are used:

$$even(0) = true \tag{9.1}$$

$$even(s(0)) = false \tag{9.2}$$

$$\forall n\ even(s(s(n))) = even(n) \tag{9.3}$$

General Proof:

	$even(n+n)$
$EXPAND\ n \rightarrow s^n(0)$	$even(s^n(0)+s^n(0))$
USE 6.2 n *times*	$even(s^n(0+s^n(0)))$
USE 6.1	$even(s^n(s^n(0)))^*$
USE 9.3 n *times*	$even(0)^{**}$
USE 9.1	**true**

Obvious choices for the cut formula here would result in circularity or a non-theorem, but the general proof gives some clues as to what is happening, and does not suggest an incorrect cut formula. Indeed, a cut formula which is suggested is that of $\forall y\ even(2.y)$, because of the way a successor function from each of the original variables pairs off together in the general proof. Notice that the sub-part of the general proof from $*$ to $**$ is linear:

$even(s^n(s^n(0)))$

$even(s^{n-1}(s^{n-1}(0)))$ by 9.3

...

$even(0)$ by 9.3

This linear structure from $even(s^n(s^n(0)))$ suggests that this is a suitable candidate for generalisation. What is happening in the general proof is that $even(s^k(s^k(0)))$ is being proved at the meta-level (the linearisation process corresponding to induction on k, as shown in the previous chapter). The generalisation suggested will therefore be $\forall k\ even(z(k))$, where $\underline{z}(k) = s^k(s^k(0))$, ie. $z(k) = 2.k$.

In a similar manner, the cut formula suggested for $\forall x\ halfint(x+x) = x$ is $\forall x\ halfint(2.x) = x$, using axioms

$$halfint(0) = 0 \tag{9.4}$$

$$halfint(s(0)) = 0 \tag{9.5}$$

$$halfint(s(s(n))) = s(halfint(n)) \tag{9.6}$$

3. $\underline{\forall x\ \forall y\ odd(x) \wedge odd(y) \rightarrow even(x+y)}$

It is necessary to add the axioms:

$$odd(0) = false \tag{9.7}$$

$$odd(s(0)) = true \tag{9.8}$$

$$odd(s(s(n)) = odd(n) \tag{9.9}$$

A cut formula is not the aim, but just the general proof:

$$\begin{array}{llll} odd(s^n(0)), & odd(s^m(0)) & \rightarrow & even(s^n(0)+s^m(0)) \\ odd(s^n(0)), & odd(s^m(0)) & \rightarrow & even(s^n(s^m(0))) \quad 6.1 \\ \downarrow & \downarrow & & \downarrow \\ strip\ off & strip\ off & & strip\ off \\ n-1\ s's & m-1\ s's & & n+m\ s's \quad 8.12 \\ \downarrow & \downarrow & & \downarrow \\ odd(s(0)) & odd(s(0)) & \rightarrow & even(0) \quad true \end{array}$$

4. $\underline{\forall x\ (even(x) \vee odd(x))}$

The nth proof depends really on whether n is even or odd. Given the ω-rule applied to $\forall x \Phi(x)$ giving subtrees (ie. general proofs of) $\Phi(n)$:

$$\frac{\begin{array}{cccc} P(even) & P(odd) & P(even) & P(odd) \\ \vdots & \vdots & \vdots & \vdots \\ \Phi(0) & \Phi(1) & \Phi(2) & \Phi(3) \end{array}}{\forall n \Phi(n)}\ \omega - rule$$

<u>General Proof:</u>

$$\begin{array}{cc} even(s^n(0)) \vee odd(s^n(0)) & even(s^n(0)) \vee odd(s^n(0)) \\ even(s^n(0)) & odd(s^n(0)) \\ even(0) & odd(s(0)) \\ true & true \\ \text{(n even)} & \text{(n not even)} \end{array}$$

When there is a disjunction in the general proof, there is a case split (see Section 6.6 above).

5. $\underline{\forall x\ even(x) \rightarrow even(x.x)}$

<u>General Proof:</u>

$$\begin{array}{lll} even(s^n(0)) & \rightarrow & even(s^n(0).s^n(0)) \\ even(s^n(0)) & \rightarrow & even(\underbrace{s^n(0)+\ldots+s^n(0)}_{n\ times}) \quad 8.10 \\ \ldots & \rightarrow & even(\underbrace{s^n(s^n\ldots(0)\ldots)}_{n\ times}) \quad \text{correct} \end{array}$$

The general proof is shown to be correct by a process of stripping off the n's analogous to that in the *even* examples above. The cut formula suggested is $\forall x\ \forall y\ even(x) \rightarrow even(x.y)$, in an analogous manner to the example 8.8 given in Subsection 8.1.5 above.

9.2 Generalisation of Constants to Variables

In this section it is considered whether rewrite steps may be saved and a more 'natural' proof produced by enabling the generalisation of constants. For example, in order to prove that $1,000,000+1 = 1+1,000,000$, the method approximating most closely to what a human mathematician would do would be to generalise the figure $1,000,000$ to a variable x, say, and then consider the problem $x+1 = 1+x$. In this way it could be seen that a variation in the figure $1,000,000$, say to $1,070,006$, would not produce a totally different proof, which is obviously what is required.

Generalisation of constants to variables arises in van der Waerden's account of the discovery of a proof of Baudet's conjecture (der Waerden, 1971). There is a strong analogy with the generalisation of terms to variables, as is shown by the prooftrees below (the first being a representation of Boyer and Moore's method of generalisation of terms to variables, and the second being an example of the sort of generalisation of constants to variables which is required).

$$\dfrac{\Gamma \vdash \forall y P(y) \quad \dfrac{\dfrac{P(f(t)) \vdash P(f(t))}{\forall y P(y) \vdash P(f(t))}\forall L \quad \forall y P(y) \vdash \forall x P(f(x))}{\forall y P(y) \vdash \forall x P(f(x))}\forall R}{\Gamma \vdash \forall x P(f(x))}\, cut(\forall y P(y))$$

$$\dfrac{\dfrac{\vdash s(0) = s(0)}{\vdash 0 + s(0) = s(0) + 0}\, 6.1, 6.2 \quad \dfrac{\dfrac{r+1 = 1+r \vdash s(s(r)) = s(s(r))}{r+1 = 1+r \vdash s(r+s(0)) = s(0+s(r))}\, hyp}{r+1 = 1+r \vdash s(r)+1 = 1+s(r)}\, 6.2}{\vdash \forall y\, y+1 = 1+y \quad *}\, cut$$

$$1,000+1 = 1+1,000$$

$$* : \dfrac{1{,}000+1=1+1{,}000 \vdash 1{,}000+1=1+1{,}000}{\forall y\, y+1=1+y \vdash 1{,}000+1=1+1{,}000}\forall L$$

Note that the proof of $\vdash 1,000+1 = 1+1,000$ is just an individual instance of the general proof of $\vdash n+1 = 1+n$. By the algorithm described above, a general proof of $n+1 = 1+n$ could be "guessed" from that of $1,000+1 = 1+1,000$ (although of course, setting $n = J$ in the proof of $P(n)$ for some number $J \neq 1,000$ might not give a correct proof). But the problem is precisely to know which constant to generalise, and not to guess a proof knowing this. More relevantly, by inspection of the proof of $1,000+1 = 1+1,000$ (given below), it may be seen that neither the 1 nor the 1,000 remain unchanged throughout the proof.

$$\begin{aligned} \vdash \underbrace{s\ldots s}_{1,000}(0)+s(0) &= s(0)+\underbrace{s\ldots s}_{1,000}(0) \\ \vdash \underbrace{s\ldots s}_{1,000}(0+s(0)) &= s(0+\underbrace{s\ldots s}_{1,000}(0)) \\ \vdash \underbrace{s\ldots s}_{1,001}(0) &= \underbrace{s\ldots s}_{1,001}(0) \end{aligned}$$

The 'guiding' heuristic will result in the unchanged terms being renamed so that they are not affected by induction. The result will be the suggestion of $s^{999}(x)+y = x+s^{999}(y)$, which is obviously not what is required. Aubin's method does not fare very well either. The expressions $s(p(s^{1,000}(0))+s(0))$ and $s(p(s(0))+s^{1,000}(0))$ are reducible and hence do not provide an ugly expression, so his method fails. Perhaps the answer is that a heuristic should be used such that large numbers which are the same on either side of the '=' sign should be generalised to the same variable, and in the case of small numbers, the same procedure for generalising terms to variables should be followed (should the latter be desired, for it would normally be large numbers for which generalisation would be of benefit).

The explanation-based generalisation methods suggests a generalisation of $\forall z\ s^{1,000}(0)+z = 1+s^{999}(z)$, which works, but is not the generalisation which is desired. However, it does capture the process of what is happening in the proof, and so perhaps should not be regarded as a failure.

In other cases, generalisation of constants to variables may be more successful. Such a generalisation in the domain of lists for $\forall l\ rev2(l,nil) = rev(l)$ was successfully carried out in Subsection 8.4.3, and is also discussed in Appendix C.3.

9.3 Generalisation on a Non-Recursive Position

If there were an example of an arithmetic proposition which required generalisation on a term which was in a non-recursive position of some function (within the proposition), this would provide an additional incentive for the proposed "guiding" method, since the latter should be able to provide a proof, whereas the other methods considered simply do not allow for generalisation on a non-recursive position (unless it is achieved by trying out every single possibility of generalisation in turn!). The type of example which is relevant is, for instance, to prove $\forall x\ even(x) \rightarrow even(x.y)$, using axioms 9.1–9.3. Ideally, it would be desirable to show (using commutativity: $x.y = y.x$) that this reduces to $even(x) \rightarrow even(y.x)$, which is easy to prove. The alternative is to allow the use of rewrite rules on a non-recursive position, namely on y in the initial expression. An alternative example where it might be necessary to generalise on a non-recursive position might occur in the type of case when there is $z.len(w)$ (perhaps nested as a substructure) on the left-hand side of an equation, and $len(w).z$, say, (nested) on the right, and one might wish to generalise $len(w)$ to y, in order to be able to apply an appropriate rewrite rule.

Such examples are too complicated for the generalisation algorithm to attempt at present, but this type of example would provide a useful domain for future work and extension of the method.

9.4 Generalisation by Inspection of the General Proof for Course of Values Induction

As has been shown in the previous chapter, so long as a general procedure for constructing a proof for each individual of a sort is specified, universal statements about objects of the sort could be proved. This section investigates whether the

generalisation method proposed would suggest a generalisation for course of values induction.

The problem may be stated as follows: given standard induction, one might wish to show

$$\forall x(\forall y(y < x \rightarrow P(y)) \rightarrow P(x)) \vdash P(z).$$

Finding the generalisation is tantamount to finding a B such that:

$$\frac{\forall x(\forall y(y < x \rightarrow P(y)) \rightarrow P(x)), B \vdash P(z) \quad \forall y(y < x \rightarrow P(y)) \rightarrow P(x)) \vdash B}{\forall x(\forall y(y < x \rightarrow P(y)) \rightarrow P(x)) \vdash P(z)} cut(B)$$

Now, define

$$Q(z) \equiv \forall x(\forall y < xP(y) \rightarrow P(x)) \vdash P(z)$$

For brevity, let $A \equiv \forall x(\forall y\ y < xP(y) \rightarrow P(x))$. The goal is to show $\forall z Q(z)$. The general procedure using the (constructive) ω-rule is to look at proofs for $Q(0), Q(1), \ldots$, then the general proof, $Q(r)$, and from the latter to construct a generalisation of Q.

The proof of $Q(0)$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, y < 0 \vdash P(y)}\, y < 0\ is\ F}{A \vdash y < 0 \rightarrow P(y)}}{A \vdash \forall y(y < 0 \rightarrow P(y))}}{A \vdash P(0)}\, x = 0\ in\ A$$

In order to prove $Q(1)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash P(0)}}{A, y < 1 \vdash P(y)}\, y = 0}{A \vdash \forall y(y < 1 \rightarrow P(y))}}{A \vdash P(1)}\, x = 1\ in\ A$$

However, the proofs do not in general display a linear proof structure. For example, the proof of $Q(2)$ is:

$$\dfrac{\dfrac{}{A \vdash P(0)} \quad \dfrac{A \vdash P(0)}{A \vdash P(1)}}{A \vdash \forall y(y < 2 \rightarrow P(y))}\, x = 2\ in\ A$$

So, it can be seen that the (succedents of the) individual proofs are of the form

$$\frac{}{P(0)}, \frac{P(0)}{P(1)}, \frac{P(0)\ \frac{P(0)}{P(1)}}{P(2)}, \ldots$$

The general proof is to prove $A \vdash P(n)$, for arbitrary n, or, in other words, $Q(n)$. This will have the form:

$$\frac{Q(0)\ldots\ \dfrac{Q(0)\ldots Q(n-2)}{Q(n-1)}}{Q(n)}$$

Note that the general proof has the same shape as the set-theoretic representation of the nth number, where $0 \equiv \emptyset$, $1 \equiv \{\emptyset\}$, $2 \equiv \{\emptyset, \{\emptyset\}\}$, $3 \equiv \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$, ..., for it can be seen that the structures are isomorphic.

In order to be able to use standard induction on the generalised goal to complete the proof-tree, it is necessary to have a linear form for the general proof of the generalisation. Therefore, the problem of generalisation may be described in terms of converting a non-linear general proof shape, such as that under consideration above, to a linear proof shape.

So, in this case, the aim is to define some $W(n)$ in terms of $Q(n)$ such that the general proof given above may reduce to a linear proof of the form:

$$\frac{\dfrac{\dfrac{W(0)}{W(1)}\,i}{\vdots}}{W(n)}\,i$$

First, note that the proof of $P(s(r))$ may be given in terms of a sub-tree Δ, plus the proof of $P(r)$ (the proof of which is Δ).

$$\frac{\Delta\ \dfrac{\Delta}{P(n)}}{P(s(n))}$$

Δ is in fact $P(0) \wedge \ldots \wedge \ldots P(n-1)$.

So, for any point in the prooftree the pattern holds:

$$\frac{P(0) \wedge \ldots P(r)}{P(0) \wedge \ldots \wedge P(r) \wedge P(s(r))}$$

This explanation is provided without using the symbol '<' at all, at this level. This information may be captured by saying that the general representation required will have proof form

$$\begin{array}{rcl} P(0) & & \\ P(0) \wedge P(1) & & \\ \vdots & & \\ P(0) \wedge \ldots \wedge P(n-1) & \equiv & \forall y(y < n \rightarrow P(y)) \\ P(0) \wedge \ldots \wedge P(n) & \equiv & \forall y(y < s(n) \rightarrow P(y)) \end{array}$$

which is a linear form.

Defining

$$W(z) \equiv \forall y(y < s(z) \rightarrow P(y))$$

then there is as required:

$$\begin{array}{c} W(0) \\ \vdots \\ \dfrac{W(n-1)}{W(n)} \end{array}$$

This would suggest that the generalised goal might be:

$$\forall x(\forall w(w < x \rightarrow P(w)) \rightarrow P(x)) \vdash \forall y(y < z \rightarrow P(y))$$

Thus, the required B at the beginning of the section is suggested as $\forall y(y < z \rightarrow P(y))$. The general proof of this generalised goal, $\Phi(z)$, follows the linear form, and hence the generalisation may be proved by induction, as would be wished. This is shown below, as the generalised goal $\Phi(s(m))$ reduces to $\Phi(m)$:

$$\begin{array}{l} \dfrac{A \vdash \forall y(y < m \rightarrow P(y))}{A \vdash \forall y(y < m \rightarrow P(y)) \wedge P(m)} \equiv \Phi(m) \\ \dfrac{A \vdash \forall y(y < m \rightarrow P(y)) \wedge P(m)}{A \vdash \forall y(y < s(m) \rightarrow P(y))} \equiv \Phi(s(m)) \end{array}$$

In conclusion, the proposed method does indeed suggest a generalisation for course of values induction, although I have no general heuristic for spotting the generalisation. Moreover, it allows a greater insight into how the general proof does suggest a generalisation, and indicates a general procedure for allowing generalisation over various domains, namely of applying proofs transformations to linearise the general proof, and then of generalising the rth case produced. The whole process which is proposed may be regarded as a new way of directing search for inductive proofs. Note that this is a similar approach to that taken for lists.

9.5 Different Induction Schemata

The general approach provided in Chapter 8 encapsulates only "simple" induction, and could be extended to cope with more advanced induction schema, if necessary. For instance, the same kind of induction as at the object level might be used, but a new successor function $\hat{s}$ defined as:[1]

$$\begin{aligned}\hat{s}^0(X) &= X\\ \hat{s}^1(X) &= s(X)\\ \hat{s}^{s(s(n))}(X) &= s(s(\hat{s}^n(X)))\end{aligned}$$

The proof of an 'even' example shall be considered: in order to obtain a linear proof, it is necessary to try to reduce the $(n+2)$nd case to the nth case. The application of rewrite rules n times will use this (as it will provide a sort of induction in order to prove n applications of the rewrite).

Using axioms 9.1–9.3, the general proof of $\forall x\ even(x+x)$ under such a schema starts:

$$\begin{aligned}&even(\underline{n+2}+\underline{n+2})\\ &even(\hat{s}^{s(s(n))}(0)+\hat{s}^{s(s(n))}(0))\\ &even(\hat{s}^{s(s(n))}(0+\hat{s}^{s(s(n))}(0)))\\ &even(\hat{s}^{s(s(n))}.\hat{s}^{s(s(n))}(0))\\ &even(\hat{s}^{s(s(n))}.(s(s(\hat{s}^n(0)))))\\ &even(s(s(\hat{s}^n.(s(s(\hat{s}^n(0)))))))\\ &even(\hat{s}^n.\hat{s}^n(0))\end{aligned}$$

In the proof, $even(\hat{s}^{s(s(n))}.\hat{s}^{s(s(n))}(0))$ reduces to $even(\hat{s}^n.\hat{s}^n(0))$, which by the linearisation process suggests that the cut formula should be $even(\hat{s}^k.\hat{s}^k(0))$, ie. $even(\hat{s}^{2.k}(0))$ (by syntactic definitions), which would translate back to give $even(2.k)$

---

[1] In this case, nested induction could be used as an alternative, since the step case $s(n)$ from the first induction would by another induction yield the base case (0, thus corresponding overall to 1), and the step case $(s(m)$, thus corresponding overall to $s(s(n)))$.

as a suggested cut formula, which works. The base case carries through fairly trivially. Note in addition that the linearisation process works here, but the linearised general proof will have the structure.

$$\dfrac{\dfrac{\dfrac{P(n)}{P(n-2)}\,i}{P(n-4)}\,i}{\begin{array}{c}\vdots\, j\\ P(0)\end{array}}$$

where $P(x) = even(x + x)$. Other structures will be analogous to the particular induction schemata used.

9.6 Nested Use of the ω-rule

For an example such as $\forall x\ \forall y (x+y)+x = x+(y+x)$, x may be converted to $s^n(0)$ and y to $s^m(0)$, say, and exactly the same rule-set used as given in the ω-proof for $\forall x\ (x + x) + x = x + (x + x)$ to reach a complete ω-proof. The generalisation suggested will be $\forall x\ \forall y'\ \forall z\ (x+y')+z = x+(y'+z)$, as with the former. However, the 'm' is not broken down in the proof at all, and so this is not a fair example of nested use. Instead the example $\forall x\ \forall y\ (x+y)+(y+x) = (x+x)+(y+y)$ shall be considered. For this example, Aubin would have to try many combinations, and J. Hesketh's method would essentially pick induction candidates in the recursive positions, suggesting a suitable generalisation:

$$\forall x\ \forall y\ \forall z\ \forall w (z + y) + (w + x) = (z + x) + (w + y)$$

The general proof is as follows:

$$\begin{array}{rcll}
\underline{(s^n(0) + s^m(0))} + \underline{(s^m(0) + s^n(0))} & = & \underline{(s^n(0) + s^n(0))} + \underline{(s^m(0) + s^m(0))} & 6.2\\
s^n(\underline{0 + s^m(0)}) + s^m(\underline{0 + s^n(0)}) & = & s^n(\underline{0 + s^n(0)}) + s^m(\underline{0 + s^m(0)}) & 6.1\\
\underline{s^n(s^m(0)) + s^m(s^n(0))} & = & \underline{s^n(s^n(0)) + s^m(s^m(0))} & 6.2\\
\underline{s^n(s^m(0) + s^m(s^n(0)))} & = & \underline{s^n(s^n(0) + s^m(s^m(0)))} & 6.2\\
s^n(\underline{s^m(0 + s^m(s^n(0)))}) & = & s^n(s^n(\underline{0 + s^m(s^m(0))})) & 6.1\\
s^n(s^m(s^m(s^n(0)))) & = & s^n(s^n(s^m(s^m(0)))) & \\
\textbf{equality} & \textit{by} & \textit{cancellation} &
\end{array}$$

It is possible to reach equality from the previous line of the general proof using cancellation. Now let us build up the most general form using explanation-based generalisation:

rule	general		form	instantiations
	$(s^n(X)+Y)*(s^m(P)+Q)$	$=$	$(s^n(W)+Z)*'(s^m(V)+U)$	**original**
6.2	$s^n(X+Y)*s^m(P+Q)$	$=$	$s^n(W+Z)*'s^m(V+U)$	
6.1	$s^n(Y)*s^m(Q)$	$=$	$s^n(Z)*'s^m(U)$	$X,P,W,V=0$
6.2	$s^n(Y+s^m(Q))$	$=$	$s^n(Z+s^m(U))$	$*,*'\equiv+$
6.2	$s^n(s^m(M+s^m(Q)))$	$=$	$s^n(s^n(N+s^m(U)))$	$Y=s^m(M),Z=s^n(N)$
6.1	$s^n(s^m(s^m(Q)))$	$=$	$s^n(s^n(s^m(U)))$	$M,N=0$
	$s^m(Q)$	$=$	$s^n(U)$	$U=s^{m-n}(Q)\ m\geq n$
	$s^m(Q)$	$=$	$s^n(s^{m-n}(Q))$	$o/w\ Q=s^{n-m}(U)$

The original expression must have been (assuming $m>n$ and $z>x$):

$$s^n(0)+s^m(0))+(s^m(0)+Q)=(s^n(0)+s^n(0))+(s^m(0)+s^{m-n}(Q))$$

which suggests a generalisation of $\forall x(x+y)+(y+z)=(x+x)+(y+(y+z-x))$ (which works with one induction).

Via the general proof, it may be seen that the problem reduces to that of $\forall x\forall y\ y+(y+x)=x+(y+y)$, but a suitable cut formula has not really been provided. There is no problem in providing a general proof, but just in providing a method of suggestion of a cut formula from the general proof.

9.7 Conclusions

This chapter has considered more speculative instances of application of the proposed generalisation than the previous chapter. In some cases the method is seen to suggest results when other current methods fail. A more detailed comparison is given in Chapter 11.

Chapter 10

Implementation

This chapter describes the proof development system Seldon used as an operating system, plus the implementation of the system $PA_{c\omega}$ (including generation of individual proofs, forming general proofs, showing correctness of the general proof, and the application of rewrite rules). In addition, a system in which the general proofs may be displayed and investigated is described, and also the implementation of the generalisation method (namely the provision of a cut formula via explanation-based generalisation methods) for simple arithmetical cases. More technical details of the implementation of the system of arithmetic with the ω-rule and of the generalisation method may be found in Appendix E, and the code is presented in (Baker, 1992).

10.1 Goals of Implementation

The thesis project divides into three main areas of interest: implementational considerations on the one hand, and formal considerations on the other (plus the application of this to generalisation), for the main goal is that the ω-rule would be implemented, and the properties of the resulting arithmetic system extended by the ω-rule investigated further. On the implementational side, it was necessary to write code which would guide theorem-proving, such that typing "`constructive_omega_rule`", for example, causes the relevant subtree of the

prooftree to be completed. Such a tactic is initially presented with the goal '$\forall x P(x)$', generates the hypotheses '$P(r)$ for uniform r', and subsequently enables this subtree to be completed. Various problems had to be overcome as to how the rule and subgoals might be represented, how to input the individual proofs $P(0), P(1), P(2), \ldots$ etc. to the system (for proof environments usually only deal with one proof at a time) and how to incorporate these proofs into the main proof. It was desirable to be able to generate these proofs, rather than inputting them, since they might not be available for input. Other implementational issues included consideration of how the constructive ω-rule would be implemented, details of the new implementational system, and in particular what a proof using the constructive ω-rule would look like. Another problem which was implemented concerns how it could be known whether the general proof is the right one — in other words, how one could know that the next individual proof example selected would not falsify it. This involves some kind of meta-level reasoning as a check to ensure that the general method is the required one (cf. Subsection 10.3.1).

10.2 Seldon

This section is intended as an overview of the Seldon proof development system. A basic acquaintance with the workings of OYSTER is necessary to understand Seldon, within which framework the programs of this project were written.

In order to carry out the implementation, a domain was required in which proofs could be manipulated. An obvious candidate was the Nuprl program development system (Constable, 1986), reimplemented in Prolog by Christian Horn, a visitor to the Mathematical Reasoning Group in Edinburgh (Horn, 1972). This system, called Oyster, is a proof checker for a version of Intuitionistic Type Theory based on that developed by P. Martin-Löf (Martin-Lof, 1970). Thus it embodies a higher-order, typed constructive logic in sequent-calculus form (see Subsection 3.2.2 for further details).

As well as being written in Prolog, Oyster is run at the Prolog prompt level. In order to introduce a theorem to work on, one may either load a previous theorem or create a new theorem. Oyster reasons backwards from the theorem to be proved, and the rules of inference are the usual rules of sequent calculus, together with mathematical induction. It is important to note that only one proof at a time may be operated upon in the system, although others may be stored and recalled later. A theorem initially consists of a sequent. The proof tree is developed by telling Oyster which rule of inference to use at each node. Some of these rules will result in new nodes being attached to the tree – others will complete the branch (giving status *complete*). One may traverse the tree by means of the commands given in Appendix B. The proof is complete when all the branches end in completed proofs. It is possible to save a proof at any stage as either an internal representation or a readable version.

The system in which the programs of this project are written is Seán Matthew's implementation of Peano and Heyting arithmetic in the Oyster framework, called Seldon[1]. This has all the advantages of a system like Oyster, but uses a much simpler logic. In addition it contains various useful tactics, and moreover dispenses with the need for copious trivial type-justifications. The behaviour of the system is very similar to that of Oyster, principally because it is constructed by replacing the object-level logic of Oyster with Peano and Heyting arithmetic. There are three sentential operators and a constant used in the propositional calculus[2]. 'A $\rightarrow \perp$' is preferred to '$\neg$ A' on the grounds that it is more intuitive in a constructive system. The rules that can be applied are those of the sequent calculus axiomatisation of first order logic given in Dummett's "Elements of Intuitionism" (Dummett, 1977, P133), together with mathematical induction. The rule of repetition is allowed, in order that the primitive recursive ω-rule may be implemented (cf. Subsection 4.3.1).

---

[1]See Table A–1 for further details of the connectives.

[2]See Table 10–1 for further details of the connectives.

Representation	*Explanation*
/\ X :: Y	for all X, Y is the case
\/ X :: Y	there exists X such that Y is the case
A => B	A implies B
A # B	A and B
A \ B	A or B
void	absurdity
A = B	A equals B
s(X)	successor of X

Table 10–1: The Connectives in Seldon

The rules of arithmetic are essentially Peano's axioms[3] plus induction. All of the axioms (except induction) are called by the basic rule, and will succeed with no subgoals if they unify against the consequent of the current sequent. In the case of induction, `induce` and `induce(X)` both generate the appropriate subgoals for an induction from a universally quantified goal. If the latter is used, the variable of the universal quantifier will be renamed to `X` in the induction cases; otherwise it will remain as given.

The search for a proof must be guided either by a human user or by a proof tactic. The latter is a Prolog program which incorporates OYSTER or Seldon commands, whichever is appropriate. The tactic may, for example, recognise if the sequent to be operated upon corresponds to a certain pattern, and in this case perform certain rules of inference upon it. It is possible to switch within Seldon from the representation of Peano Arithmetic to that of its intuitionistic counterpart, Heyting Arithmetic. Therefore this allows the representation of either $PA_{c\omega}$ or $HA_{c\omega}$; $PA_{c\omega}$ is normally used as there are more desirable theoretical results for this system (see Section 2.5).

Each proof is built up in the form of a tree, and every stage of the tree may be

[3]See Section 4.1.

displayed on the screen with information as to the hypotheses, goals, position in the tree and whether the subtree is proved below it. A typical such position is shown below, as shown on the screen by the *display* command, which displays the current node of the prooftree. This partial proof corresponds to the application of the rule $\rightarrow -r$ to $\vdash a \vee \neg a \rightarrow (\neg\neg a \rightarrow a)$; comments on the right-hand side have been added afterwards.

```
                                          %%% Status - partial
t : [] partial autotactic(idtac)          %%% Top of tree
==> (a\a=>void)=>((a=>void)=>void)=>a     %%% sequent to be proved
by rimp                                   %%% rule used

  [1] incomplete                          %%% the only subgoal
  v0. a\a=>void                           %%% the hypothesis
  ==> ((a=>void)=>void)=>a                %%% the goal
  by _                                    %%% next rule not yet
                                          %%% given
```

In order to gain the flexibility of classical proof techniques without losing the advantages that a constructive proof offers, it would be desirable to have both the option of a classical and of a constructive system. This is done by introducing the law of excluded middle:

$$\frac{\Gamma \vdash (A \rightarrow \bot) \rightarrow \bot \; ext \; A_{ext}}{\Gamma \vdash A \; ext \; because(A_{ext})}$$

which may in fact be optionally included in the system by the user.

For further details about the system, see Appendix E.

10.2.1 Extract Terms

As might be expected from the underlying philosophy,[4] constructive mathematics is motivated by the conviction that "there exists" means, or rather should mean, "we can find explicitly". The view of one famous proponent, Bishop, is that if

[4]See Subsection 2.1.1.

mathematics is properly written, one should be able to extract what he called "numerical information" from the proof (Bishop, 1967).[5] This may be regarded as a program, which contains the computational information implicit in the proof. Thus $\vdash G\ ext\ G_{ext}$ is a proof of G, where G_{ext} is a member of the type identified with G, called the *extract term* of G. The extract term is also known as a *witness term* for G, because it is the object whose existence provides a proof of G. In Seldon, in order to be able to refer to hypotheses, labels are used, so that sequents will have the form:

$a : A, \ldots, y : Y \vdash Z\ ext\ Z_{ext}$

where $a, \ldots, y$ are the labels and may be free in Z_{ext}.

For each rule of inference there is a constructor which links together the constructions corresponding to the arguments (subgoals) of the major connective the rule involves. The extract term is the constructor applied to the appropriate arguments. The extract terms may be executed by application to arguments and symbolic evaluation, in other words they themselves may be run as a program to perform the function described by the proof.

In extending the system to arithmetic, rather than providing Peano's axioms with complicated witness terms as appropriate, the witness terms in the rules other than the induction axiom are replaced with the simple 'axiom'. This does not affect the computational part of the generated witness terms, and merely makes the witnesses easier to read. For example,

$$\frac{}{\Gamma \vdash a + 0 = a\ \ ext\ \ axiom}$$

The particular case of the induction rule is as follows:

$$\frac{\Gamma \vdash A(0)\ \ ext\ \ Base \qquad \Gamma \vdash \forall x.A(x) \rightarrow A(s(x))\ \ ext\ \ Step}{\Gamma \vdash \forall x.A(x)\ \ ext\ \ \lambda x.induce(x, Base, Step)}$$

where

---

[5]Bishop did not use type theory, however.

$induce(0, b, s) = b$; **(base case)**

$induce(s(n), b, i) = i(n)(induce(n, b, s))$. **(induction step)**

As a way of explaining the above notation, $factorial(x)$ could be defined as `induce(x,s(0),lambda(a,lambda(b,s(a)*b)))`, where `lambda(a,b)` is the code used to represent the lambda term $\lambda a.b$. This expression represents the defining equations '$factorial(0) = s(0)$' and '$factorial(s(n)) = s(n).factorial(n)$'.

So, what is the extract term corresponding to the ω-rule? Yoccoz's result about the equivalence of encoding with recursive trees (Yoccoz, 1989a) discussed in Section 2.5 has implications relating to the 'justification' or 'extraction' associated with the ω-rule, and the consequences for program synthesis. For a proof in Heyting arithmetic (HA)[6], we will have

$$\frac{P(0)\ ext\ e(0) \ldots P(k)\ ext\ e(k) \ldots}{\forall x P(x)\ ext\ \lambda n.e(n)} \omega - rule$$

If the prooftrees $P(i)$ are generated in some uniform manner, there is no problem, because $e(i)$ will be dependent upon i in a uniform way, and the extract may be calculated as just being $\lambda k.e(k)$. However, if the prooftrees are *not* generated, there will not be a uniform function e of n. The standard way of obtaining an extract $e(i)$ from $P(i)$ may be used, but it is not obvious how to obtain the extract term for the denominator. One possible solution, with reference to Subsection 2.1.2 above, is that the prooftrees may be converted to a canonical form, and thus a general pattern calculated. Note also that it would be possible to store information about the structure of the individual proofs for the sequents in the numerator of the ω-rule by means of extract terms.

[6]The intuitionistic counterpart of PA. See Section 4.1 for further details.

10.3 Implementational Representation of General Proofs

In this section I consider how the general proofs described in Chapter 6 are represented from an implementational point of view. The basic implementational approach has already been briefly presented in Subsection 6.1.2. For an application of the ω-rule, a description of the nth subproof, uniformly with respect to the parameter n, is required. An example representation, in the linear case, for the general proof for $P(n)$ is:

$$gen([R1(Pos1, f1'(n)), R2(Pos2, f2'(n)), \ldots]).$$

This represents the application of each rewrite rule $RJ\ fJ'(n)$ times at subposition $PosJ$ of the resulting formula of the general proof. For example, this might be $gen([rule1([], n), rule2([2], 3), rule1([1], 2 * n)])$. The representation for each individual proof for $P(J)$ would be of the form:

$$proof(J, [Rule1_J(Pos1_J, T1_J), Rule2_J(Pos1_J, T2_J), \ldots])$$

This is a predicate, its first argument being the natural number J, and the other argument being an ordered list of predicates such that $RuleK_J$ is the Kth rule to be applied in the proof, and is applied TK_J times, with $PosK_J$ specifying the position at which it is applied. Note that we may have $RuleA_J = RuleB_J$.

In order to *automatically* recognise a general pattern, and hence form the basis for an implementation, upon input of individual proofs for $P(1), P(2), \ldots$, it is necessary to provide an algorithm to do this.[7] (However, in this thesis it is the underlying structure of the general proof which we wish to exploit, rather than the way in which the general proof is obtained.)

The algorithm which is automated in the code (see Appendix E.5) and which performs the required function is described below. Essentially, it generalises an

---

[7] cf. Inductive inference (Section 3.3).

initial individual proof, and then updates this generalisation according to other individual proof examples until the general proof representation satisfies all of the (large number of) cases considered.

First pick some proof of $P(J)$. In general it is best to have J as some fairly low number, but not too low or the proof might exhibit some individual variance. For instance, the proofs for 0 or 1 are nearly always special cases.

For each $P(J)$ as it is considered, guess the various $hK_N(n)$ such that $hK_N(J) = TN_J$, and store these in the form

$$list(J, RuleN_J(PosN_J, TN_J), [h1_N(n), \ldots hr_N(n)]).$$

Each $hK_N(n)$ is a guess of how to generalise TN_J with respect to J, ordered such that hJ is more likely to be correct than hK, if $J < K$. Some sort of heuristics would have to be applied to enable choice of the most likely functions. For example, products could be preferred to sums, which were preferred to powers, so that upon input of $J = 2$ and $T1_J = 4$, the machine would 'guess' $h1_N(X) = 2X$, then $h2_N(X) = X + 2$, then $h3_N(X) = X^2$, say, and so on.

The next stage is to rewrite the individual representation of the proof $P(J)$ to that of a general proof by setting $fN = h1_N$, ie.

$$gen([Rule1(Pos1, f1(n)), Rule2(Pos2, f2(n)), \ldots])(\star)$$

where $fK(J) = TK_J \ \forall K.(\Delta)$
This is the initial guess of the general proof from one proof example (and hence may well not be the correct guess).

Then inspect the representation of the next individual proof (ie. $P(K)$, where $K = J + 1$), which will be of the form:

$$proof(K, [Rule1_K(Pos1_K, T1_K), Rule2_K(Pos2_K, T2_K), \ldots]$$

Compare this with $(\star)$ to see if the general proof guess should be revised:

1. If $Rule1 = Rule1_K$ and $Pos1 = Pos_K$ and

(a) $f1(K) = T1_K$, set $f1' = f1$ (ie. the guess $f1$ looks as if it might be correct) and move on to consider the second predicates in the lists.

(b) (i) If $T1_J = T1_K$, (ie. $T1_{J+1}$) then $T1_J$ must be a number, since it is not dependent on J. So assert the following in place of the previous statement: '$list(K, Rule1_K(Pos1_K, T1_K), [T1_K])$'. Set $T1' = T1_K$, and by the above other guesses will not be allowed.

(ii) If $T1_J \neq T1_K$, recurse to try other possibilities at (Δ). Note that the functions we originally chose could be the wrong ones, and this stage is an indication of that fact. What will be done at this stage is to remove $f1(n)$ from the stored list of guesses for the Nth list (ie. to assert instead '$list(N, Rule1_K(Pos1_{K,}), [L - f1(n)])$'. Then $\forall M \leq N$ inspect '$list(M, Rule1_K(Pos1_{K,}), L_M)$' and pick Z such that $Z \in L_M, \forall M$. Set $T1' = Rule1_K(Pos1_K, Z)$.

2. If $Rule1 \neq Rule1_K$,

   find h st. $h(K) = T1_K$.

   If $h(N) = 0$ for N<K, then add $Rule1_K(Pos1_K, h(n))$ to the general list.

   Otherwise, something has gone wrong.

Continue in this manner, comparing the two lists and extending the general form where necessary. Repeat with the new general form and the next particular proof (of $P(J + 2)$), and so on. After a finitely large number of examples, if there is a general pattern to the proofs, then it will be seen to emerge and will be encapsulated by the final representation of the general proof. In the code there is a check such that if the general proof has not been altered a set number of times when undergoing comparison with particular proofs, the process will terminate and output the general proof. Note that this general proof is still only a guess, and needs to be checked, as described in the next section.

In practice, this algorithm needs modification due to the fact that some rules may be applied in a different order and yet have the same effect, so that proofs which do not have the same surface structure could yet have an identical general proof.

There would have to be some check made at the beginning as to whether the rules of the proofs could be permuted in order to give the same end result, and whether this would indeed provide a general proof. In particular, problems are caused because the automatic generation of individual proofs described in Appendix E.4 may result in the rewrite rules needing to be permuted before the algorithm above may be applied. The example considered in Appendix E.4.2 demonstrates that account must also be taken of the fact that there are alternative forms of the notation, in the sense that $rule([1,1],2)$ may be equivalent to $rule([1,1],1)$ followed by $rule([1,1,1],1)$ for example. Hence, the algorithm described above depends heavily on how the individual proofs have been generated: it will only work if they have been generated uniformly (cf. `make_thms2` of Appendix E.5); if not, a more advanced algorithm is required. Given as input individual proofs for which the common structure is already apparent and which do not require internal rearrangement within each proof, the implementation is able to provide a guess in polynomial time for the general proof for any arithmetical examples for which the appropriate rewrite rules are defined in advance within the system: examples (8.1) to (8.8) from Table 8.1 illustrate the type of examples involved.

The algorithm described in this subsection is needed in order to be able to procede to the next stage of implementation; however, as discussed above, the one used is not particularly clever, and could be replaced instead by any appropriate inductive inference algorithm (such as Plotkin's least general generalisation (Plotkin, 1969)). In general, the main body of work is done at this stage because the complexity of the algorithm needed to guess a general proof from non-uniformly generated examples is exponential, whereas the stages to be described in the following sections of checking the general proof and suggesting a cut formula are only polynomially complex, and this is reflected in the time taken to produce the result. In conclusion, the algorithm described above needs a fair amount of modification to work satisfactorily on relatively simple examples if they do not already show a common structure, although it works straightforwardly on examples which do, and in general one is forced to rely heavily on others' attempts to show that the inductive inference process may succeed: such as the work of Rouveirol, who has tackled

the problem of controlling the hypothesis generation process to get only the most relevant candidates (Rouveirol, 1990). As an alternative, the user may bypass this whole stage by specifying the general proof directly.

10.3.1 Checking for Correctness of the General Proof

This subsection describes the implementational algorithm corresponding to the process of justifying derived rewrite rules via meta-induction, which has been discussed in Section 6.4. Note that the algorithms described in both this and the previous section have been implemented (see Appendix E for further details). The "checking" stage of implementation deals with the problem of how it could be established that the general 'proof' is indeed a proof of the formula of interest — in other words, how it could be known that a selected individual proof example could not falsify it. This necessarily involves some kind of meta-level reasoning as a check to ensure that the proposed general rule applications do indeed give a proper proof when applied to the general case of the sequent to be proved. Thus, there is a need for some sort of checker separate from Seldon to be developed. The whole process should not be confused with inductive inference, that is the action of building up the general proof from individual proof instances, which is the concern of the previous section.

Initial Solution

The general idea is to take the nth case, and apply the rules in the manner indicated by the general proof, in the hope of reaching equality. For example, if one were trying to prove $\forall x(x+x)+x = x+(x+x)$ by means of the constructive ω-rule, and the general proof was of the form

$$gen([rule1([1,1],n), rule1([2],n), rule2([2,2,1,1],1), rule2([2,2,2],1), rule1([1],n)])$$

(which had been inferred from individual instances), then the method of checking would be along the lines of that indicated below.

Problem

There is a problem as to how it is actually possible to carry out the checking detailed above within the framework of a computer program. In particular, how is it possible to apply a rule n times, when the value of n is not known? If a rule transformation is applied repeatedly, it will not be known when to stop. Without having a numerical value for n, which by the nature of n is not possible, this is what will happen:

$$\begin{array}{rl} s(s(s^{n-2}(0))) & \\ \vdots & \textit{rule } R \textit{ repeated} \\ s\ldots s(s^{n-123}(0))) & \\ \vdots & \textit{rule } R \textit{ repeated} \end{array}$$

It is not possible just to apply the rule a few times, and then guess the general form of the rule after a random transformation, because this does not constitute a proof and besides is the stage which is being justified.

Revised Solution

As an example, let us imagine we are trying to prove $\forall x\ x + 3 = 3 + x$ by means of the constructive ω-rule, and a proposed general proof has been obtained via the previous stages of the implementation. A random case is inspected for the number n', namely $n' + 3 = 3 + n'$.

1. Substitute $n' \rightarrow s^n(0)$, and similarly for other numbers. The result in this case is $s^n(0) + s^3(0) = s^3(0) + s^n(0)$.

2. Recall that the general solution is stored in terms of rules such as "apply R n times at position P". Try applying such a rule a few times in order to *guess* the general pattern in terms of m, the number of times the rule is applied. In other words, from $A(0)$, use the rule to get $A(1), A(2), A(3), \ldots$ and then substitute m for these variables in the resulting expression (where $\ldots A(i) \ldots$ is the sequence of formulae in the general proof).

So,

$$\begin{array}{rrcll} & A(1) & \equiv & s(s^{n-1}(0)+y) & \\ & A(2) & \equiv & s(s(s^{n-2}(0)+y)) & \equiv \quad s^2(s^{n-2}(0)+y) \\ & A(3) & \equiv & s(s(s(s^{n-3}(0)+y))) & \equiv \quad s^3(s^{n-3}(0)+y) \\ GUESS & A(m) & \equiv & s^m(s^{n-m}(0)+y) & \end{array}$$

Actually, it might be that a number was actually a constant throughout all cases, and so we would not wish to substitute m for it. Hence a list of possible guesses should be made for each $A(r)$, and the intersection of these given as $A(m)$.

Note: We are defining $A(m)$ for all $m \leq n$. If we had $m = n+1$, $A(m)$ would be $s^{n+1}(s^{-1}(0)+y)$, which would not be obtainable using rules (1) and (2) because these rules do not introduce s^x with negative x. A similar argument applies for any other $m > n$.

3. Now, prove this guess using meta-induction. If this does not work, move back to Step Two and try again with a more complicated $A(m)$, dependent on a larger number of base examples.

   The fact that $A(0)$, the original formula, may be transformed by rewrite rules to $A(m)$, for any m, may be represented as:

$$\frac{A(0) \quad A(r) \vdash_{transformation} A(s(r))}{A(m)}$$

   For the example above, we have:

$$\frac{s^n(0)+y^{(1)} \quad s^r(s^{n-r}(0)+y) \vdash s^{r+1}(s^{n-(r+1)}(0)+y)^{(2)}}{s^m(s^{n-m}(0)+y)^{(3)}}$$

   where (1) is the original (sub)formula under consideration, (2) is to be proved and (3) is the result of Step Two above.

   To prove (2), take $A(s(r))$, apply R in reverse, and tidy up the expression using definitions such as $s^n(0) = s(s^{n-1}(0))$, to give $A(r)$.

$$ie.\ s^{r+1}(s^{n-(r+1)}(0)+y) \Longrightarrow_{Rrev} s^r(s^{n-r}(0)+y)$$

4. We have now proved that $A(0)$ may be transformed by rewrite rules to $A(m)$, for any $m \leq n$. In particular, we are interested in the case when $m = n$. We have then proved that $A(0)$ is transformable to $A[m/n]$ — in this case that $s^n(0) + y$ reduces to $s^n(0 + y)$.

5. Substitute $s^n(y)$ for $s^n(0+y)$ by using rule (2). If $f(n)$ is constant, as in this case, carry out a straight substitution, otherwise follow steps two onwards for the particular rule given by the general proof. Substitute $f(n)$ for m at step four. Continue in this manner, recursing along the members of the rule-list of the general proof.

   Substitution of the transformation given by the last rule provided by the general proof should result in an axiom of the form $X = Y$, where Y is reducible to X by various syntax definitions (viz. when the rule-list is empty). If it does not, then the general proof is not a correct one as there must be cases in which it could be falsified.

In conclusion, meta-induction was used to prove the guess for the new line of the general proof, as discussed in Subsection 6.4.1.

10.4 System Description (cf. $PA_{c\omega}$)

The implementation is carried out within the framework of an interactive theorem-prover with Prolog as the tactic language, in which the object-level logic is replaced by classical and constructive theories of arithmetic. Any finite number of individual instances of proofs of a proposition may be generated automatically by the use of various tactics. The general representation of the proofs is provided by having an initial generalisation from a 'starting' proof, and then by updating this general proof using the other individual proofs, until the general proof seems to have reached a stable form. This general proof is then checked to see if it is indeed the correct one, as described above. The whole of this stage has been automated.

There are two options which are allowable from the Seldon proof (in PA or HA) with goal $\Gamma \vdash \forall x P(x)$. One is to ask to use the constructive ω-rule, whereby the system will check to see whether there is a correct general proof, and then return to the former system and close the subtree, or else state that the application of the rule failed. The user may then continue to investigate other positions in the prooftree. The other option is to ask for an appropriate cut to be carried out in PA (the cut being worked out by the system from the general proof), with a further option to complete the tree as far as possible (using standard theorem-proving techniques). The general proof may be provided automatically, but there is an option in each case to switch temporarily to another system which will allow for the description, manipulation and display of the general proof. The user may specify the proof incrementally, in terms of applications in positions in the tree, plus induction over a distinguished parameter, or all at once — and this is checked. The system builds up a recursive function description of the general proof, and is able to display individual proofs in addition to the general case. A cut formula is automatically suggested from general proofs using an implementation based on the method of explanation-based generalisation. This whole process is summarised in Appendix E.8.2.

Branching proofs have been discussed in Section 6.6. A tree representation is included in the implementation, and hence extra branching code is not needed.

Transcripts of the system running are given in the Appendices. See Appendix E for technical details of the implementational representation of the ω-rule. For further details of the implementation, and the code written, see (Baker, 1992).

In summary, code has been written for:

- representation of $PA_{c\omega}$. This involves implementation to:
  - represent the ω-rule (and subgoals).
  - produce or input the individual proofs.

- allow interaction with user to define general proof recursively, or else produce the general proof automatically. This involves application of rewrite rules, and a tree representation.

- check correctness of the general proof.

- generalisation: suggestion of cut formula (only arithmetical case implemented).

- implementation of cut elimination — see (Baker, 1990).

10.5 Summary

The implementation was carried out within the framework of an interactive theorem-prover with Prolog as the tactic language, in which the object-level logic is replaced by classical and constructive theories of arithmetic. The implementation allows both the automatic or incremental construction of ω-proofs, and the validations of descriptions of ω-proofs.

Chapter 11

Comparison with Related Work

"Men are more apt to be mistaken in their generalisations than in their particular observations."

Niccolò Machiavelli

This chapter presents first of all relevant work for which no direct comparison will be made, namely reflection and proof transformation, and then moves on to provide a comparison of relevant contemporary approaches with the methods proposed in this thesis. The following section introduces so-called reflective systems, and compares these with the closely-related semi-formal systems including ω-rules, which have been considered in Section 2.5. Next, in Section 11.2, an analogous process to the linearisation of the general proof described in Section 8.3 is considered, namely that of proof transformation. Note that related work in the fields of inductive generalisation and explanation-based learning has already been presented in Sections 3.3 and 3.4, and that in these areas the work described in this thesis does not offer more highly developed algorithms, such as to carry out inductive inference, but rather provides a new domain for these methods (see Section 3.4 for details).

The main field which is a candidate for comparison with existent methods is the generalisation method suggested in Chapters 8 and 9; this is contrasted in Section 11.4 with the alternative methods available, which are discussed in Section 11.3. There is little work being carried out which is analogous to the process

described in the thesis of one logical system guiding the provision of cut formulae in another, and so this provides a new approach.

11.1 Related Theoretical Work: Reflection

In this section systems of reflection will be considered, since there is a strong parallel between rules of reflection and ω-rules. Generally, reflection principles extend a system with a statement of confidence about that system: by reflecting upon the meaning of the system, it might be able to code insight into further true mathematical statements that were not deducible beforehand. As was discussed in Chapter 2, Penrose has put forward the argument that reflection, in the sense of insight, allows "natural reasoners" (ie. mathematicians) to outperform formal systems (Penrose, 1989). Gödel writes in the opening paragraph of (Gödel, 1958):

> "An extension of finitism by such concepts was explicitly suggested by Bernays in (Bernays, 1983, P258–271). By abstract concepts, in this context, are meant concepts which are essentially of the second or higher level, ie. which do not have as their content properties or relations of *concrete objects* (such as combinations of symbols), but rather of *thought structures* or *thought contents* (eg. proofs, meaningful propositions, and so on), where in the proofs of propositions about these mental objects insights are needed which are not derived from a reflection upon the combinatorial (space-time) properties of the symbols representing them, but rather from a reflection upon the *meanings* involved."

An example of such a reflection principle would be, for a given system $\mathcal{T}$, an assertion about the soundness of $\mathcal{T}$ of the form: "For every formula A in the language of $\mathcal{T}$, if A is provable in $\mathcal{T}$, then A is true". However, for any theory $\mathcal{T}$ to which Tarksi's theorem on the undefinability of truth is applicable, a truth predicate is not definable in the language of $\mathcal{T}$ (Feferman, 1962). The notion that whatever is provable in $\mathcal{T}$ is true may be conveyed without use of the formal notion of truth by what is called the local reflection principle (for each sentence A

of the language, where $\ulcorner\ \urcorner$ denotes Gödel encoding[1] and $P_{\mathcal{T}}$ is a formal provability predicate for the system $\mathcal{T}$):[2]

$$\exists y\ P_{\mathcal{T}}(y, \ulcorner A \urcorner) \rightarrow A$$

This makes sense for any theory $\mathcal{T}$ containing S_0, where the latter is defined as in (Feferman, 1991, P10).[3] Indeed, Turing was the first to use '$Prf_{\mathcal{T}}(\ulcorner A \urcorner) \rightarrow A$' to achieve completeness for all sentences with only universally quantified variables in arithmetic. This work was extended by Feferman, who put forward the *uniform reflection principle*, one form of which is

$$\forall n(\exists y\ P_{\mathcal{T}}(y, subst(\ulcorner A \urcorner, n)) \rightarrow A(n)) \qquad (RP)$$

where $subst(\ulcorner A \urcorner, n)$ is the Gödel number of the sentence obtained by substituting $\underline{n}$, the numeral representing the number n, for x in $A(x)$. Feferman used this to achieve a partial completeness result, in the sense that every proposition that is true in the standard model is provable eventually in one of the systems obtained from iterating this extension. For Π_1-statements local reflection and uniform reflection over PA are equivalent, but generally the latter is stronger than the former (Feferman, 1962). Kreisel and Lévy have shown that PA extended with the rule RP is equivalent to PA extended with TI_{ϵ_0}, where TI_{ϵ_0} is transfinite induction up to ϵ_0 (definable by $\forall n\ \forall m\ (m \prec n \rightarrow \Phi(m)) \rightarrow \forall n\ \Phi(n)$, where $\prec$ is a well-ordering on the natural numbers of ordinal type ϵ_0) (Kreisel & Lévy, 1968). This result is a good illustration of how addition of a reflection principle allows generalisation to a universally quantified sentence from individual cases (which is also the effect

---

[1] See Section 2.4 for discussion of Gödel numbers.

[2] Gödel's system has both the provability predicate $P_{\mathcal{T}}(m, n)$, which is defined as holding if and only if in the system $\mathcal{T}$, m is the Gödel number of a proof of the formula with Gödel number n, and the proof predicate $Prf_{\mathcal{T}}(n)$, which holds if and only if n is the Gödel number of a proof in the system $\mathcal{T}$.

[3] In other words, any theory $\mathcal{T}$ strong enough to formally represent the basic syntactical notions for arbitrary effectively generated axiomatic theories, and to derive the basic properties concerning them.

of the ω-rule); in this case the individual cases are provided by Gentzen's proof that induction up to η is provable in PA, for any particular ordinal η less than ϵ_0.

From this it may be seen that reflection systems are closely related to ω-rules. In general, the former add to a system statements of the form "if there is a formal prooftree, then the statement S should hold", and are more efficient because the meta-level is at least partly represented within the system. However, they do not explicitly represent prooftrees, but rather deal with provability in a system. ω-rules, on the other hand, allow an explicit representation of infinite proofs, and allow manipulation of individual proofs, rather than concentrating on provability. So there is a proof predicate in the systems with the ω-rule, as opposed to the provability predicate in the reflective systems.

Hence with the ω-rule it is possible to examine individual proofs and, in particular, to think in terms of producing a generalisation, which is a useful application of the work presented in this thesis.[4] In addition, as may be seen from a comparison of completeness results of the various systems,[5] the addition of an ω-rule may well produce a stronger system than the addition of a corresponding reflection rule.

Several systems have been developed which implement reflection rules. Perhaps the most famous is the system FOL, which is an interactive system developed by R. Weyhrauch at Stanford (Weyhrauch, 1980). It implements first order logic via a modified form of natural deduction. Meta-theory is used in the manipulation of algebraic expressions in a way which is not explicitly used by most other theorem-proving systems: the system has a separate meta-theory, which is not given in the form of Gödel encoding. Reflection is enabled by a theory of knowledge represented via reflection principles. The automation of the system is very restricted: only certain limited types of deduction may be made automatically, although very complicated ones may be guided by a user. The use of meta-theoretic knowledge can shorten proofs at the object-level, and the interac-

---

[4]See Chapter 8.

[5]See also Section 2.5.

tion between meta- and object-level theory may increase the power and efficiency of the theorem-prover. Weyhrauch and Aiello have also carried out related work to show how the use of meta-theoretic knowledge results in improving the quality of various algebraic proofs, making them both easier to find and to understand (Aiello & Weyhrauch, 1980).

Kowalski has developed an alternative system which effectively mixes the meta- and object-level (Bowen & Kowalski, 1982). The object-level language is extended to cover the part of the metalanguage which deals with the former's provability relation. This is intended to overcome the problem of database management, due in particular to changing databases. By amalgamating the object-language and part of the meta-language it is possible to explicitly refer to theories and discuss derivability from these theories. Rules from the meta- to the object-language are linked by pairs of reflection principles. The main difference between this work and Wehyrauch's is that the latter does not completely amalgamate object- and meta-language. More recently, S. Matthew at Edinburgh University has also implemented a reflective system (Matthews *et al*, 1991).

As discussed above, although there are strong parallels between such systems and the implementation described in this thesis, the latter is a more suitable method both for manipulating and for generalising individual proofs.

11.2 Transformation of Proofs

The approach of linearising the general proof to obtain another general proof from which the cut formula might be obtained is analogous to work being carried out to linearise programs with more than one recursive call. In particular, P. Madden has optimised recursive programs by substituting induction schemata related to corresponding synthesis proofs (Madden, 1989). For example, starting with a course of values inductive proof to produce the Fibonacci sequence $1, 1, 2, 3, 5, 8, 13, \ldots$ represented by the algorithms

$$fib(0) = 1 \tag{11.1}$$

$$fib(1) = 1 \tag{11.2}$$

$$\forall x\ fib(s(s(x))) = fib(s(x)) + fib(x), \tag{11.3}$$

in order to calculate $fib(n)$, two recursive calls (on $fib(n-1)$ and $fib(n-2)$) are made, which in their turn do the same. Hence the computational complexity is exponential. Madden transforms this proof into one corresponding to the algorithm (where <> denotes the pair constructor)

$$newfn(0) =< 1, 1 >$$

$$newfn(n+1) =< v_1 + v_2, v_1 > \text{ where } newfn(n) =< v_1, v_2 >$$

The values of the two step cases of the less efficient algorithm above have been combined (within fn), so that the function $newfn$ requires only n recursive calls (to $newfn(0)$), and hence the computational tree resulting from this type of induction is linear. It can be seen that this work is related to the linearisation process described in Section 8.3, in the sense that the solution of the $n+1$st case is achieved by reducing this to the nth case.

Other related work in the field of program and proof transformation includes Burstall and Darlington's system for transforming programs which are expressed as recursion equations (Burstall & Darlington, 1977), and W.N. Chin's doctoral work (Chin, 1990).

The following sections directly compare the generalisation method presented in Chapters 8 and 9 with current alternatives.

11.3 Methods of Generalisation

The generalisation method proposed in Chapter 8 makes it relevant to assess current generalisation methods. The major generalisation heuristics are given below: systems such as INKA and CLAM[6], borrow heuristics from Boyer and Moore

[6]See Section 3.2.

and Aubin. A more thorough overview of current generalisation techniques may be found in (Hummel, 1987).

11.3.1 Boyer and Moore's Heuristics

In order to ascertain its usefulness, the method of generalisation proposed in this thesis should be compared with current methods. Of these, perhaps the most famous is that implemented by Boyer and Moore,[7] and described in Section 3.2, in which the main heuristic for generalisation is that identical terms occurring on both the left and right side of an equation are picked for rewriting as a new variable (with certain restrictions) (Boyer & Moore, 1979). This may be a quick method if it happens to work, but may also entail the proofs of many lemmata, which might need to be stored in advance in anticipation of such an event in order to be more efficient. The proof is not 'tailor-made' to the axioms given in the sense that the proof provided by the guiding method would be, and may in general be considered an 'ad hoc' approach. Moreover, this method will not always work, notably when the generalisation of variables apart is required, and overgeneralisation is also possible. The method for generalisation used by the CLAM system (see Section 3.2) is based on the same heuristic as that of NQTHM, and thus the same criticisms may be levelled against it.

11.3.2 Aubin's Method

The problems inherent in Boyer and Moore's approach led Raymond Aubin to develop their work (Aubin, 1975). Aubin's PhD thesis deals with the question of generalisation, and in particular how one might generalise a formula so that it would be provable by induction.

---

[7]It is possible for the user to turn the generalisation heuristic off for Boyer and Moore's system. The heuristic does not rely upon anticipation of any future induction which may be needed (cf. Section 9.3).

In order to explain Aubin's method in more detail, it is first necessary to explain what is meant by "symbolic evaluation", "ugly expressions" and "destructor functions", and give some background to his work. This was heavily influenced by the approach of his supervisors, Boyer and Moore. Although the famous theorem-prover of the latter was written in LISP, whereas Aubin's thesis is implemented in POP-2, Aubin's work is essentially a specialised development of part of their work, in the sense that the positive information value of the failure of symbolic evaluation is used to even greater effect. Using a LISP-like language including the function *cond*, it is not necessary to use multiple equations when making recursive definitions. ($Cond(x, y, z)$ is defined as follows: if $x = nil$ then z else y.) The function *append* (viz. that function which joins two lists) may be defined recursively as follows:

$$append(X, Y) = cond(X, cons(car(X), append(cdr(X), Y)), Y).$$

Symbolic evaluation is essentially the rewriting of an expression using a set of (rewrite) rules. Given the axioms that

$$car(cons(X, Y)) = X$$

$$cdr(cons(X, Y)) = Y$$

if an expression of the form "$append(A, B)$" is (symbolically) evaluated by means of the definition above, then if A is of the form '$cons(X, Y)$', or if A is nil, the *car* and *cdr* may be eliminated. If it is not, there will not be rules which will eliminate the *car* and the *cdr*, and so it will not be possible to eliminate '*append*'. In this case A will be called an "ugly expression". Generally, if a rewriting threatens to produce an ugly expression then the rewriting is prohibited and symbolic evaluation is terminated.

Boyer and Moore's theorem prover was able to use the ugly expressions as a guide to choosing the correct induction scheme for a particular expression. However, in a case such as proving

$$append(rev(A), append(B, C)) = append(append(rev(A), B), C))$$

their theorem-prover would fail, basically because the induction hypothesis is not strong enough to support the induction conclusion. This example may however be generalised to the associativity of append, which may be proved fairly straightforwardly, by replacing the term $rev(A)$ with a new variable, say D. Induction would then be carried out on D. Aubin stressed that in fact the term to be replaced should be an ugly expression. The procedure is that there must be the same ugly expression produced from either side of the equation, and if so, this expression is replaced by a new variable and then induction is carried out on this new variable. The exact mechanisms will become clearer when going through worked examples later in this chapter.

One major problem which occurs is that of over-generalisation (the generalisation of a theorem into a non-theorem). For example,

$$length(length(X)) = length(X)$$

where $length(X)$ is defined as taking an argument X of type list, and returning a list of *nils* (representing a number denoting the length of the list)[8]. One of the occurrences of $length(X)$ becomes trapped as an ugly expression during the symbolic evaluation of $length(length(X))$. If both occurrences of the term are replaced by a new variable, the result is no longer a theorem, because $length(Y) = Y$ is true only when Y is a list of *nil*s. Boyer and Moore's solution was to add extra information, in this case of the form "if Y is a list of *nil*s, then $length(Y) = Y$". Aubin preferred to avoid over-generalisation by noting which occurrences of the ugly expression (ie. $length(X)$ in this case) are trapped, and allow only these terms to be replaced by a variable, rather than all the equivalent ones. In this example, there is only one trapped ugly expression, so generalisation should not be allowed on the grounds that there is no trapped ugly expression equivalent to this on the other side of the equation.

In summary, Aubin's method is to "guess" a generalisation by generalising occurrences of variables in the argument positions of functions in which the recursive

[8]See definitions given by axioms 8.17 and 8.18.

cases are defined, and then to work through a number of individual cases to see if the guess seems to work. If it does work, he will look for a proof. If it does not, then he will "guess" a different generalisation. J. Castaing adopts a very similar approach: she replaces, by universally quantified variables, all those positions in the induction hypothesis which do not match with those of the induction conclusion, and then identifies those variables which replaced identical terms and were primary recursion variables, separately identifying all the others (Castaing, 1985).

Example One

As an introductory example, we will consider how Aubin's method would produce a generalisation for '$\forall l\ append(l, append(l, l)) = append(append(l, l), l)$'. His whole approach is summed up on pages 4-5 of his thesis;

> Firstly, how is it possible to generalize only certain occurrences of a term in a goal? For example, can we get
>
> $$app(k, app(l, l)) = app(app(k, l), l)$$
>
> from
>
> $$app(l, app(l, l)) = app(app(l, l), l)?$$
>
> Answering this question requires a deeper understanding of generalization than what can be found in Boyer and Moore, or Brotz: they generalize by replacing all occurrences of a term by a new variable. My solution derives from thinking of generalization and selection of induction variables as two facets of the same problem. That is, I generalize only those term occurrences which would be suitable to do induction upon had they been variables. Grossly speaking, a variable is a suitable candidate for induction if it is the first variable which a call-by-need[9] interpreter tries to evaluate (unsuccessfully) in the induction goal. With this method, I can earmark the first and fourth occurrences of l in the above problem, and then, generalize only these to a new variable k.

---

[9] Aubin is describing the process by which rules are applied during symbolic evaluation: the interpreter will consider what it needs to know in order to evaluate the whole term, then what it needs to know in order to evaluate that, and so on recursively. This is the call-by-need part.

Symbolic evaluation on the original equation would result in four basic expansions of append, as outlined below (the original equation is written in this instance as "$append(l_a, append(l_b, l_c)) = append(append(l_d, l_e), l_f)$" to clarify the positions of the particular 'l' under discussion):

$append(l_a, l_b) = cond(l_a, cons(car(l_a), append(cdr(\underline{l_a}), l_b)), l_b).$
$append(l_b, l_c) = cond(l_b, cons(car(l_b), append(cdr(\underline{l_b}), l_c)), l_c).$
$append(l_c, \ldots) = cond(l_c, \ldots append(cdr(\underline{l_c}), \ldots), \ldots).$
$append(append(l_c, l_d), l_c) =$
$cond(append(l_c, l_d), cons(\ldots, append(cdr(\underline{(append(l_c, l_d)}, \ldots), l_c).$

The ugly expressions are underlined — note that they are the argument to a destructor function (in this case, '*cdr*') such that the complex structure containing them cannot be broken down by rewrite rules, since there are none applicable.

It can be seen that '$append(l_c, l_d)$' is not suitable to be generalised as an induction variable, since only one of its occurrences would be unpacked by symbolic evaluation. Aubin stresses this by essentially giving the condition that the induction candidate should both be one of the ugly expressions and have resulted from both sides of the equality as an ugly expression. In this case the only terms to fit this description are 'l_a' and 'l_d', or 'l_b' and 'l_d'. Aubin chooses the first pair, since these appear in recursive argument positions, and hence would be evaluated first by a call-by-need interpreter. Hence, the 'l' in the first and fourth positions should be generalised into a new variable, say k, and then induction should be done on k. So the generalisation for the original formula is

$$\forall l \forall k \; append(k, append(l, l)) = append(append(k, l), l).$$

This is the method Aubin proposes, and the induction does indeed work. My treatment of this example has been given in Section 8.4, in which the more general generalisation of $\forall l \forall k \forall m \; append(k, append(l, m)) = append(append(k, l), m)$ is suggested.

Example Two

I now move on to consider an arithmetic example, namely $\forall x\ x+(x+x) = (x+x)+x$. This has strong parallels with Example One. In Peano arithmetic, the role of lists in the previous example is played by the natural numbers $0, s(0), s(s(0)), \ldots$, etc. The sucessor function, s, plays the part of the constructor function, *cons*, and the predecessor function, p, plays the part of the destructors, *car* and *cdr*, where

$$p(s(X)) = X \quad \textit{and} \quad p(0) = 0.$$

An analogous function to *cond* is defined, namely *cond2* such that $cond2(X, Y, Z)$ represents "if $X = 0$, then Z, else Y". Thus, addition (with X as the induction variable) may be defined as:

$$X + Y = cond2(X, s(p(X) + Y), Y)$$

So, to return to the example under consideration, symbolic evaluation using the definition of plus would result in the following sub-expressions (with positions clarified as before):

$$x_i + \ldots = cond2(x_i, s(p(x_i) + \ldots), \ldots)$$

where i is a, b, or d (or indeed, x_i is $x_d + x_e$, but this is ruled out for generalisation since it occurs only once).

The ugly expressions here will hence be x_a, x_b, and x_d, since these are the arguments of the destructor function which cannot be broken down. Again taking the left-most, the first and fourth occurrence of x will be generalised into a new variable, y, giving:

$$\forall x \forall y\ y + (x + x) = (y + x) + x$$

Induction is then done on y. Note that this is essentially the same generalisation as given by the simplified version of the guiding method, where structure remaining the same in the general proof is renamed. The guiding method for this example is given in Subsection 8.1.3.

Example Three

In addition, Aubin describes a technique for generalising terms with a constant in an accumulator position, namely his indirect generalisation method (Aubin, 1976), which is applicable for recursive formulations of iterative computations such as the factorial representation $it\_fac(n, m) \Rightarrow$ if $n = 0$ then m; if $n \neq 0$ then $it\_fac(p(n), n.m)$. The local variable m stores the intermediate results of the computation, ie. it "accumulates" what has been computed so far, and is therefore called an accumulator variable. Iterative programs use accumulator variables which are usually initialised with specific values (eg. $m = 1$) before the loop is entered.

However, Aubin's approach requires the use of lemmata, such as $fac(n) = fac(n).1$. From $\forall n fac(n) = it\_fac(n, 1)$ he replaces $fac(n)$ by $fac(n).1$ and then replaces 1 (the initialisation of the accumulator) on both sides of the equation by a new variable m. Thus, he suggests a generalisation of $\forall n \forall m fac(n).m = it\_fac(n, m)$ from $\forall n fac(n) = it\_fac(n, 1)$. This is equivalent to inverted application of replacement and substitution rule (cf. Subsection 3.1.2). See also (Hesketh, 1991b, P39–41) for an analysis of his method with regard to the *rev*2 example (cf. Appendix C.3).

11.3.3 Hesketh's Approach

Aubin's work has been extended by Jane Hesketh, for example to include the introduction of accumulators to solve certain generalisations which had hitherto not been possible (Hesketh, 1991b). Rippling paths generalise Aubin's notion of recursion paths (in the sense that other sorts of rewrite rules may be used). Essentially the method she advocates is to look at the proof of $P(s(x))$ and to try to use the available axioms to reduce this to $P(x)$; generalisation is directed by the failure of heuristics employed in proof planning. Hesketh has extended CLAM to deal with generalisation apart in this manner (Hesketh, 1991b), but getting a solution necessarily involves a fair amount of heuristic knowledge and the testing of alternative proposed generalisations. However, she does have access to extra information over

and above the method proposed in Chapter 8, such as access to a strong sense of directionality which the rippling heuristic provides (Hesketh, 1991b).

11.3.4 Summary

The main methods of generalisation have been presented in this section. In summary, Boyer and Moore only generalise common terms to variables (and are liable to overgeneralise), while Aubin concentrates on this (via generalising variables apart), plus the generalisation of terms with a constant in an accumulator position. First of all Aubin will try to use induction upon variables in the primary recursion position (that is, those on which a function is recursively defined). If that fails he will try other recursive argument positions. However, there are many possibilities, and so he has to try out some instantiation of variables to give individual cases in the hope of finding a counterexample which would refute that particular generalisation. Aubin's method is partially theoretically justified — he will pick out at least some of the 'problem cases', but not necessarily all of them, and easily too many, by his method of finding ugly expressions. Hence he has to have some method to guess the pairs when he has too many (he chooses the left-most ones), and in the case when there are too few, the method fails. If the arguments in primary recursion positions fail as induction candidates, as it would with the example

$$\forall x \;\; x.(x + x) = (x.x) + (x.x)$$

where it would suggest that the first and fourth x's should be renamed, he would consider others occurring in recursive argument positions, but not primary recursive positions, such as the second and sixth x in the above equation. However, the number of combinations could grow fairly quickly, and so he used a program which instantiated some values for the variables and tested them in order to detect false generalisations rapidly. Hence his approach is not terribly rigorous, does not always work, and moreover involves a fair amount of trial and error. His method is a very 'ad hoc' approach, and a theoretical one with a higher success rate would be preferable. In particular, if a constructor such as a successor function appears in

Methods	Types of Generalisation			
	vars apart	*terms to vars*	*constants*	*adding accumulators*
Boyer & Moore		X		
Aubin	X		X	
Hesketh	X		X	X
Guiding Method	X		X	X

Table 11–1: Scope of Applicability of Generalisation Methods

an original goal, together with individual variables, Aubin's method may result in over-generalisation or indeed no solution at all. Jane Hesketh's approach succeeds in many cases, is more flexible than Aubin's, and does not suffer from the same problems, but nevertheless obtaining a solution often involves trying out several alternative possibilities.

Table 11–1 gives an overview of the types of generalisation targetted by the various methods. A comparison of the methods outlined in this section and the generalisation method suggested in Chapter 8 will be given in the following section.

11.4 Comparison of Different Generalisation Methods

In this section various examples are considered with reference to the generalisation methods discussed above, and an overview of the guiding method is given.

11.4.1 Worked Examples

Generalisation of Terms to Variables: $x + s(x) = s(x) + x$.

Would Aubin's method be able to produce a correct solution for an example which included a successor function in the original equation, or would he over-generalise? Let us consider the example above, for which a correct solution was provided by

the 'ω-guiding' method. Rewriting the original equation $x_a + s(x_b) = s(x_c) + x_d$ by means of the definition for plus, we have

$$s(p(x_a) + s(x_b)) = \ldots$$

and

$$\ldots = s(p(s(x_c)) + x_d)$$

The latter reduces to $s(x+x)$ using the relevant axiom, and so there is no trapped ugly expression from the right hand side of the equation. There is an ugly expression produced on the left hand side, namely x. However, Aubin's method will not produce a solution here, because there is an ugly expression only on one side of the equation. In general, Aubin is unable to generalise expressions which would require a generalisation of a variable which is present in the equation both individually and as an argument to a function (giving a term), for example when x and $s(x)$ both need to be generalised to y and $s(y)$, in order to do induction on y.

Note that NQTHM would adopt the following strategy for this example: it would look for common subexpressions, and hence replace $s(x)$ by y on both sides of the equation, giving the commutativity of plus as the goal to be proved. This would be a valid approach, but it might well be that insufficient axioms were available to be able to prove the commutativity of plus; this is a more general generalisation, and so this method is liable to overgeneralise. Moreover, the generalisation $x+y = y+x$ does not give the solution immediately, as shown below — the tree requires another induction in order to close.

$$\dfrac{\dfrac{\begin{array}{c}\textit{Complete}\\ \vdots\\ x+z=z+x \vdash x+s(x)=s(x)+x\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \vdash 0+z=z+0\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\ \textit{doesn't close}\\ r+z=z+r \vdash s(z+r)=z+s(r)\end{array}}{r+z=z+r \vdash s(r+z)=z+s(r)}\,hyp}{r+z=z+r \vdash s(r)+z=z+s(r)}\,6.2}{\vdash x+z=z+x}\,ind(x)}{\vdash x+s(x)=s(x)+x}\,cut(x+z=z+x)$$

Hence, the guiding method might be considered a more direct approach, and the suggested cut formula of $x + s(y) = s(x) + y$ a preferable one (since it works with one induction).

Jane Hesketh's approach would label all the x occurrences and hence the wave-fronts (ie. structure given by rewrite rules — see (Hesketh, 1991b)) around them as potentially changeable, but it would be the second and fourth occurrences which would be blocked. The generalisation suggested probably would also be $x+s(y) = s(x)+y$, since this would be the direct generalisation of the blocked parts, remembering that it would be starting from the failure of $s(x + s(s(x))) = s(x + s(x))$, and would be seeking to clear the second and fourth blocked wavefronts.

Generalisation Apart: $x.(x + x) = x.x + x.x$

The guiding method for generalisation of this example has been considered in Subsection 8.5.1, and carries through. Aubin is able to produce a correct solution after an incorrect one (for further details of the proof, see (Aubin, 1975, Appendix 3)), but this involves a fair amount of trial and error. Jane Hesketh has tried to improve upon his attempt, but could not get the example to work easily either. See however (Hesketh, 1991a) for a partial solution to $\forall x\ x.(x + x) = x.x + x.x$, inspired by the information provided by the failed method of reduction of $P(s(x))$ to $P(x)$. Essentially the problem with the latter approach is that guesswork, heuristics and trial and error must be involved to a certain extent in finding a solution.

Next, a further, more complicated, example, namely $x \neq 0 \rightarrow p(x) + s(s(x)) = s(x) + x$, will be considered in detail below, and a comparison of the various methods available will be given.

Destructor Terms: $x \neq 0 \rightarrow p(x) + s(s(x)) = s(x) + x$

In this section we compare and contrast the various methods of generalisation which might be expected to tackle a proof such as $x \neq 0 \rightarrow p(x) + s(s(x)) = s(x) + x$. This example is an interesting one due to the fact that its proof must involve at least one cut, that it is a conditional proposition and further, that both constructor and destructor functions are involved. This poses problems for the

major generalisation methods. The general proof, and subsequent cut formula suggested by the 'guiding method' is given above in Subsection 8.1.5.

The Boyer-Moore Theorem Prover

The Boyer-Moore Theorem Prover fails when asked to prove this particular example, using only the recursion equations (cf. 6.1, 6.2, 8.11 and 8.12). The transcript returned by the theorem prover (NQTHM) is given in Appendix D. Note that NQTHM uses a lemma to replace X by SUC Y in order to remove PRED, but then detects looping, in the sense of the new equation being a degenerate form of the original one. It then fails, as there is nothing to induct upon. In other words, we have the reduction from

$$x \neq 0 \rightarrow p(x) + s(s(x)) = s(x) + x$$

to:

$$x \neq 0 \rightarrow p(x) + s(s(x)) = s(s(p(x) + x))$$

and then setting $x \Rightarrow s(z)$ to eliminate $p(x)$, to give

$$z + s(s(s(z))) = s(s(z + s(z)))$$

which is no advance on the original.

In fact, straightforward generalisation of common subexpressions would be expected to suggest a cut formula of $\forall x\ x \neq 0 \rightarrow p(x) + s(z) = z + x$. However, induction on x fails for this expression, too. A simplified form of this (partial) proof which eliminates the conditions and includes only the step part of the proof is given below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\ldots \vdash r + s(s(t)) = s(r + s(t))}{r + s(t) = t + s(r) \vdash r + s(s(t)) = s(t + s(r))}\,hyp}{r + s(t) = t + s(r) \vdash r + s(t) = s(t) + s(r)}\,6.2}{\forall z\ r \neq 0 \rightarrow p(r) + s(z) = z + r \vdash r + s(z) = z + s(r)}\,ind(z)}{\forall z\ r \neq 0 \rightarrow p(r) + s(z) = z + r \vdash p(s(r)) + s(z) = z + s(r)}\,8.12}{\forall x\ \forall z\ x \neq 0 \rightarrow p(x) + s(z) = z + x}\,ind(x)$$

Hence the CLAM system is not able to provide a solution to this problem either.

Aubin

By Aubin's method, using the definition

$$X + Y = cond2(X, s(p(X) + Y), Y)^{10}$$

one obtains

$$s(p(\underline{p(x)}) + s(s(x))) = \ldots$$

and

$$\ldots = s(p(s(x)) + x)$$

The ugly expressions are underlined. The second expression reduces to $\ldots = s(x+x)$, and so there is no ugly expression for the right hand side of the equation. Aubin's method requires that there should be the same ugly expression on each side of the equation before generalisation may take place, and hence his method fails.

The destructor would not cause a problem for Jane Hesketh's method, because her technique would just use whatever recursion equations were available. The method would start by experimenting with eliminating the blocked wavefronts; due to the presence of axiom 8.12, the $p(x)$ would be left and the second and fourth occurrences of x generalised to y, thus providing the generalisation $x \neq 0 \rightarrow p(x) + s(s(y)) = s(x) + y$.

Addition of Accumulators: $\forall l.\ len(rev(l)) = len(l)$

Note that this can also be proven using weak fertilisation followed by a generalisation (NQTHM-style), as shown in Appendix C.1.

The general proof is as follows:

---

[10]Note that this definition is equivalent to the definitions $0 + y = y$ (ie. 6.1) and $s(z) + y = s(p(s(z)) + y)$, so by using 8.12 we obtain, $s(z) + y = s(z + y)$, (ie. 6.2). Hence this definition, plus 8.11 and 8.12, is equivalent to the axioms 6.1, 6.2, 8.11 and 8.12 above. *(cond2(X,Y,Z)* represents "if X=0, then Z, else Y".)

$$
\begin{array}{rcl}
len(rev(\mathcal{L}(n))) & = & len(\mathcal{L}(n)) \\
len(rev(x_{s^{n-1}(0)} :: \mathcal{L}(n-1)) & = & len(x_{s^{n-1}(0)} :: \mathcal{L}(n-1)) \\
 & & \textit{by } \mathcal{L} \textit{ def} \\
len(rev(\mathcal{L}(n-1)) <> x_{s^{n-1}(0)} :: nil) & = & \ldots \\
 & & \textit{by } 8.18, 8.21 \\
len(rev(x_{s^{n-2}(0)} :: \mathcal{L}(n-2)) <> x_{s^{n-1}(0)} :: nil) & = & \ldots \\
 & & \textit{by } \mathcal{L} \textit{ def} \\
 & \vdots & \textit{similarly, } n \textit{ times} \\
len((rev(x_0 :: \mathcal{L}(0)) <> x_{s(0)} :: nil) <> \ldots x_{s^{n-1}(0)} :: nil) & = & \ldots \\
len(x_0 :: nil <> x_{s(0)} :: nil <> \ldots <> x_{s^{n-1}(0)} :: nil) & = & \ldots \\
 & & \textit{by } 8.21,\ 8.13,\ 8.14 \\
len(x_0 :: x_{s(0)} :: \ldots :: x_{s^{n-1}(0)} :: nil) & = & len(x_{s^{n-1}(0)} :: \ldots x_0 :: nil) \\
s^n(0) & = & s^n(0) \\
 & & \textit{by } 8.18 \\
 & & EQUALITY
\end{array}
$$

$R(s(n)) \equiv len(rev(\mathcal{L}(s(n))) = len(\mathcal{L}(s(n)))$

and

$R'(n) \equiv len(rev(\mathcal{L}(n)) <> x_{s^n(0)} :: nil) = len(x_{s^n(0)} :: \mathcal{L}(n))$

$= len((x_{s^n(0)} :: nil) <> \mathcal{L}(n))$

$R(0) \equiv len(((rev(\mathcal{L}(0)) <> (x_0 :: nil)) <> x_{s(0)} :: nil) <> \ldots <> x_{s^{n-1}(0)} :: nil)$

$= len(x_{s^{n-1}(0)} :: \ldots :: \mathcal{L}(0))$

ie. $len(rev(\mathcal{L}(0))) <> (x_0 :: x_{s(0)} :: \ldots x_{s^{n-1}(0)} :: nil)$

$= len(rev(x_0 :: x_{s(0)} :: \ldots :: x_{s^{n-1}(0)}) <> \mathcal{L}(0))$

This suggests the generalisation

$$len(rev(l) <> a) = len(rev(a) <> l)$$

Note the difference between this suggestion provided by the guiding method, and the suggestion which would be obtainable by Hesketh's method[11] (Hesketh, 1991b) of:

$$len(rev(l) <> a) = len(a <> l)$$

The generalisation suggested by the guiding method is preferable, as it reveals more structure, and may be proved using only one induction, rather than two, as shown in Appendix C.1.

---

[11] This example is not actually considered by Jane Hesketh, but this is the generalisation which it would produce.

Further Examples: $\forall x\ even(x + x)$; $x.(y.0) = y.(y.0)$

There are also examples which demonstrate how the guiding method may provide information which other methods do not: cf. the $even(x + x)$ example in Section 9.1. The Boyer-Moore strategy does not apply here. Obvious choices for the cut formula here would result in circularity or a non-theorem, but the general proof gives some clues as to what is happening, and does not suggest an incorrect cut formula. Indeed, the cut formula which is suggested is that of $even(2.y)$ (cf. Section 9.1). This is a useful improvement on the alternative methods. Other such examples are discussed in Subsection 8.1.5. Another example which illustrates the advantages of the proposed method of generalisation has been presented in Section 9.3. A comparison of the approaches relating to the generalisation of examples involving lists has already been given in Subsection 8.2.2.

For the example $x.(y.0) = y.(y.0)$, NQTHM gets lost in an infinite loop. Basically, what happens is that an induction on x is chosen. The induction hypothesis takes the form $x.(y.0) = y.(y.0)$ while the induction conclusion takes the form $s(x).(y.0) = y.(y.0)$ (note that the NQTHM destructor-style proof has here been converted into constructor style). Performing a rewrite application on the conclusion gives: $(x.(y.0)) + (y.0) = y.(y.0)$ and fertilisation (left-right) gives $(y.(y.0)) + (y.0) = y.(y.0)$. Then the fatal generalisation (namely substituting z for $y.0$) is performed, to give: $(y.z) + z = y.z$. This process is repeated infinitely. The guiding method is not able to provide a cut formula, but a general proof is given. Such a pattern may be exploited by the user to suggest a cut formula, even though the generalisation algorithm is not sophisticated enough to do this.

By working through the examples above, hopefully it has been demonstrated that there is a justification for the use of this method, at the very least for a particular type of example. Note that the generalisation method suggested in Chapter 8 may still be useful experimentally even if it breaks down, to a greater extent than any of the alternative methods, and that it also provides a uniform approach for generalisation.

Methods	Selection of Types of Example		
	$x.(x+x) = x.x + x.x$		$(x+x)+x = x+(x+x)$
Boyer & Moore	*fail*		$y+x = x+y$
Aubin	$x.(y+y) = x.y + x.y$	1.	$(x+y)+y = x+(y+y)$
Hesketh	$x.(y+y) = x.y + x.y$	2.	$(x+y)+y = x+(y+y)$
Guiding Method	$x.(y+z) = x.y + x.z$		$(x+y)+z = x+(y+z)$
	$x + s(x) = s(x) + x$		$even(x+x)$
Boyer & Moore	$x+y = y+x$		*fail*
Aubin	*fail*		*fail*
Hesketh	$x + s(y) = s(x) + y$	3.	*fail*
Guiding Method	$x + s(y) = s(x) + y$		$even(2.x)$ 4.
	$x \neq 0 \rightarrow p(x) + s(s(y)) = s(x) + y$		$rotate(len(l), l) = l$
Boyer & Moore	*fail*		*fail*
Aubin	*fail*		*fail*
Hesketh	$x \neq 0 \rightarrow p(x) + s(s(y)) = s(x) + y$	5.	$rotate(len(l), l <> a)$
Guiding Method	$x \neq 0 \rightarrow p(x) + s(s(y)) = s(x) + y$		$rotate(len(l), l <> a)$
	$rev2(l, nil) = rev(l)$		$x.(y.0) = y.(y.0)$
Boyer & Moore	*fail*		*fail*
Aubin	$rev2(l, a) = rev(l) <> a$		*fail*
Hesketh	$rev2(l, a) = rev(l) <> a$		*fail*
Guiding Method	$rev2(l, a) = rev(l) <> a$		*general proof only* 6.

1. Solution achieved by testing various possibilities after initial failure of method.
2. Solution achieved by pruning search space.
3. Probable result (this example was not considered in (Hesketh, 1991b)).
4. Generalisation algorithm fails, but this cut formula is 'suggested' from the general proof.
5. Probable result (this example was not considered in (Hesketh, 1991b)).
6. Generalisation algorithm fails, but user might be able to suggest a generalisation from the general proof.

Table 11–2: Cut Formulae Suggested Using Different Generalisation Methods

11.4.2 Overview of the Guiding Method

The structure of a proof according to given defining recursion equations is revealed by the general proof, and it is not necessary to store or prove large numbers of sub-lemmata as Boyer and Moore do. The general proof may be seen as revealing the *structure* of the proof. The structure which remains constant throughout the general proof will not be a candidate for providing induction variables, and more specifically, the use of explanation-based generalisation from the general proof will produce the most general generalisation of the original expression which does not involve additional structure. The problem with which Aubin is faced, namely candidates for induction appearing on one side only of the equation, will not occur, and so this will not result in a failure of the method. Formal justification for the claim that the cut formula suggested will be a correct one may be provided by standard explanation-based generalisation arguments, or else the linearisation comparison from Section 8.3 in the case when additional structure (in the form of the introduction of accumulators) is involved.

11.4.3 What are the Types of Generalisation for which the Guiding Method is Advantageous?

- The type of examples for which the guiding method works and others probably do not are examples which are not solved simply by replacing similar terms on each side, and in which a constructor such as a successor function appears in an original goal, together with individual variables (and particularly if destructor functions are also involved) eg. $x \neq 0 \rightarrow p(x) + s(s(x)) = x + s(x)$.

- More speculative examples, such as those given in Chapter 9. On such examples, if all existing methods of generalisation fail, at least the guiding method does provide some helpful information for the user.

- Examples involving added structure such as a on one side of the equation, and $rev(a)$ on the other (cf. accumulator examples).

See Table 11–2 for a comparison of how the various methods fare with regard to particular examples, and Table 11–1 for a comparative analysis of the generalisation methods considered in this chapter.

11.5 Conclusions

In the theorem-proving community, there is much current interest in generalisation. The almost universal approach has been to consider the goal and try to generalise it, but very little has been done from the viewpoint of looking at individual proofs and generalising these. As shown in this chapter, this latter approach has many advantages and can work when other methods do not. The proposed guiding method provides a uniform approach which captures in many cases what the alternative methods can do (in some cases with less work), plus it works on examples on which they fail. It does not have to check extra criteria, nor do model-checking on candidate generalisations, and there is less likelihood of overgeneralisation to a non-theorem.

Chapter 12

Conclusions and Further Work

In this chapter suggestions for further work are given, together with a summary of useful contributions achieved by means of the work described in the thesis.

12.1 Further Work

Suggested further work divides into three main categories: an extension of the implementation of arithmetic with the ω-rule; incorporation with existing work within the field of theorem-proving; and thirdly, an extension of the generalisation procedure suggested in Chapter 8, both implementationally and theoretically. These extensions will be further discussed below.

12.1.1 Implementation of $PA_{c\omega}$

There are several ways in which the implementation of $PA_{c\omega}$ could be extended. Although the rules and axioms of the system are implemented, the implementation given could be extended, say to encompass giving ordinal assignments to arithmetical derivations, and therefore bounds on lengths of proofs (in the manner indicated in, say (Buchholz & Wainer, 1987)). In addition, further automation within this system could be provided, either by means of automatic completion of proofs (by writing or interfacing with an existing system such as CLAM) or by means of writing programs which prove various properties within this system. Examples of the

latter might be implementation of cut elimination for $PA_{c\omega}$, for which the foundational work has already been carried out as mentioned in Subsection 2.3.5 (and discussed in more detail in (Baker, 1990)), or a conversion of proofs in $PA_{c\omega}$ to proofs in $HA_{c\omega}$ (in an analogous manner to the conversion of classical proofs in PA to constructive proofs in HA presented in (Baker, 1989)).

12.1.2 Wider Use of the System

It would also be of benefit to reimplement those parts of the implementation which would prove useful for other users (such as some of the generalisation heuristics) within a more widely used system, in order to incorporate such ideas with current work in other areas.

My present work is implemented on a system with very few users, as the main concern was the results, rather than the system. Yet a tactic for generalisation which worked in cases when others did not might be useful to many people in automated deduction. Therefore, it would make sense to implement this using a more popular theorem-prover. This would result in a wider justification for the work, in that it would provide a useful tool for other people engaged in automated reasoning. Some of the latter would be involved in practical applications of formal methods, such as program and hardware synthesis. Research could be carried out within a generic proof development environment such as Isabelle (Paulson, 1986), which would provide a sophisticated environment for developing theorem proving strategies.

In addition, the link between inductionless induction and the generalisation approach presented in this thesis could be further investigated, for Muffy Thomas's work in finding similarities in critical pairs forms an analogue of ω-proofs (see Subsection 3.2.4). An analysis of inductive completion with respect to rippling may be found in (Barnett *et al*, 1992), but there is scope for such a comparison to be further elucidated, although at present it is not at all obvious how this may be done.

12.1.3 Extension of Generalisation Method

The extension of the generalisation work described in Chapters 8 and 9 falls into the two main categories of extending the theory suggested therein, and extending the implementation which has already been carried out.

It is hoped that by using an approach which is substantially different from previous work in this field, cut formulae might be suggested which hitherto were not possible by existing methods. The work described in this thesis justifies this claim, especially within the domains of natural numbers and lists. However, the approach applies more generally to other inductively defined datatypes. The wider application of this approach would provide a better generalisation tactic, which would be of benefit to other people using theorem-proving systems, regardless of their motive in doing so. Hence this work would have repercussions in other, diverse fields.

Further work could cover two main issues:

- The further development of a theory of validation of general proofs, which would encompass many different domains.

- The mechanised application of this theory to performing generalisations via a suitable automated deduction formalism. In fact, one could extend a general purpose theorem prover to include a generalisation tactic. Such an implementation will be of use:

  - to people involved in automated theorem-proving, particularly if the cut elimination theorem does not hold for the logical system in which they are working.

  - as a basis for further investigation of the merits of this method of generalisation as opposed to other methods, and also of the generalised theory proposed.

- as a basis for a co-operative theorem-proving environment, in order to exploit useful information given by the general proof whenever the generalisation algorithm fails.

These issues are considered below.

New Systems

The work described in the thesis concerns mainly the domains of natural numbers and lists. A further canonical example to consider would be that of (binary) trees. It would be interesting to extend the generalisation algorithm and heuristics described in Chapter 8 to arbitrary datatypes. It would be hoped that by looking at many different types of domains and systems, a general theory could be seen to emerge, and that this could be encapsulated in the implementation, which should be generic in this sense. The approach would be taken of automatically generating the general representation of an object of a certain type from the system definitions, and then automatically producing a general proof and using this to guide proofs in the system.

It would be very interesting and worthwhile to develop this approach and apply it to a much broader range of domains, and then to implement this work. The result would be a theorem-prover which would succeed in producing correct generalisations in certain cases when others cannot, and might well be faster, more direct and involve less search space. As such it would prove useful for other people involved in theorem-proving, quite apart from any theoretical results produced from such an investigation.

Development of a Theory of General Proofs

The more general question addressed by this work may be described as follows: supposing the objects of a data-type may be enumerated as $O_1, O_2, O_3, \ldots$, we wish to produce a proof of the general case $P(O_r)$, in order to prove $\forall x\ P(x)$. The step from the individual proofs $P(O_1)$, $P(O_2)$, etc. to the general proof via

the ω-rule has analogies across various data structures. A useful generalisation for theorem-proving would facilitate induction proofs in the sense of producing a suitable generalisation P' of P such that P' could be proved by induction. Thus, the whole process which is proposed may be regarded as a new way of directing search for inductive proofs. Note also that some of the individual proofs may involve the use of induction, while in other systems they may also involve use of the cut rule.

Further Types of Generalisation

Most of the examples already considered in the thesis involve only generalisation apart, or else generalisation of common subexpressions; in these cases the method works fairly straightforwardly. It would be desirable to clarify the algorithm for the cases considered in Chapter 9, which "suggest" generalisations, and extend it to other types of generalisation such as generalisation of constants to variables (cf. Section 9.2), and in addition to extend the method to cover existential statements within constructive arithmetic, for which the general proofs would provide a summary of their computational content. It is envisaged that in such cases a useful improvement on current methods could be produced.

Extension of the Generalisation Implementation

Alterations to the implementation of the generalisation method proposed could be the automation of the generalisation methods suggested for lists, and the implementation of conditional rewrite rules. Longer-term goals would be automation of the generalisation method applied to other datatypes, as discussed above.

12.2 Conclusions

The following comprises a summary of the contribution of the thesis:

12.2.1 $PA_{c\omega}$

In this section the value and rôle of the implemented system $PA_{c\omega}$ is assessed, together with the related contributions to proof theory.

With the goal of automatic derivation of proofs within some formalisation of arithmetic in mind, a version of arithmetic with the ω-rule has been developed that is suitable for use in automated reasoning. It has been put into a form in which it could be implemented, thereby realising the potential of the ω-rule in a computational setting. The motivation for doing this is given in Section 2.7.

Hence, it is possible to carry out automated theorem-proving in the system of arithmetic with a form of the ω-rule. This is a system which other people might use for doing proofs (ie. a proof environment). It is possible also to use automation to implement metatheorems within this representation of arithmetic, such as cut elimination. Moreover, certain expressions not provable in the normal formalisation of arithmetic without cut (or indeed not provable in PA itself) are provable in this system (cf. Subsection 7.2.2).

When a general proof is available, or proofs of instances can be found easily, it may be easier to use this approach than to search directly for a proof in PA. Furthermore, in practice there is no problem of generalisation in this system.

In addition, there are the broader repercussions of these points with respect to the contribution to automated deduction (see Section 1.2), by which the work constitutes an aid to automated deduction.

12.2.2 Generalisation

In the following discussion the contribution of the thesis in the field of generalisation is considered.

An application of implementation of the system of arithmetic with the ω-rule is a new method of generalisation by means of "guiding proofs" (see Chapters 8 and 9), which sometimes succeeds when other methods fail. This is especially

important as generalisation is a major theorem-proving problem (cf. Section 3.1). Moreover, by means of the method described in Chapter 8 which involves use of explanation-based generalisation in a new domain, the most general generalisation of an expression is achieved, in a uniform manner; Jane Hesketh writes in her thesis that this has not been done before (Hesketh, 1991b, P271) — for her generalisation method

> "The generalisation from
>
> $$\forall x.x + (x + x) = (x + x) + x$$
>
> will be found as
>
> $$\forall x \forall y.x + (y + y) = (x + y) + y$$
>
> not
>
> $$\forall x \forall y \forall z.x + (y + z) = (x + y) + z$$
>
> but I know of no theorem prover that can find this last generalisation in a principled way, ie. other than with just trial and error."

By the method suggested in Chapter 8, the generalisations are found by recognising patterns, and a uniform approach for generalisation is provided.

Note also how if an ω-proof is provided, even without a generalisation being suggested, something has still been achieved, in the sense that a pattern might still emerge for the user. Thus the method may still be useful within a co-operative environment if it breaks down (as shown by the examples considered in Chapter 9). This contrasts with alternative methods of generalisation, which do not provide much information if they fail. Moreover, because the suggested method explicitly exploits general patterns, it has a higher-level structure and thus greater potential for extension than other more special-purpose approaches.

This method of generalisation may be automated (and has been for simple arithmetic examples — see Appendix E.8.3). The method works for examples for which others do, plus some new examples (see Section 11.4). There is also the possibility of extending it to arbitrary datatypes (Section 8.2).

12.2.3 Implementation

The following have been carried out:

- implementation of a system of arithmetic with the ω-rule, which involves inductive theorem proving. This work has been implemented in a way which allows interaction with the user and the construction of a primitive recursive general proof.
- implementation of the generalisation method suggested in Chapter 8 for simple arithmetical cases, by means of the automatic suggestion of an appropriate cut formula.

12.2.4 Final Conclusions

A representation of arithmetic with the ω-rule has been proposed, and implemented in a way which allows interaction with the user, and the construction and manipulation of an effective (ie. primitive recursive) general proof. When a general proof is available, or proofs of instances can be found easily, it may be easier to use this approach than to search directly for a proof in PA. Furthermore, a new, uniform approach to generalisation is proposed by means of the ω-proof "guiding" a proof in PA, which may succeed in providing a valid proof in PA when other methods of generalisation fail. This general approach works for theories other than arithmetic and logics other than a sequent version of the predicate calculus, and may rather be regarded as suggesting a general framework. So long as a general procedure for constructing a proof for each individual of a sort is specified, universal statements about objects of the sort could be proved. Thus it appears that the approach described in this thesis may be an aid to automated deduction, and provides a mechanism for guiding proofs in more conventional systems.

References

Aiello, L. and Weyhrauch, R.W. (1980). Using meta-theoretic reasoning to do algebra. In Bibel, W. and Kowalski, R., (eds.), *5th Conference on Automated Deduction*, pages 1–13. Springer Verlag. Lecture Notes in Computer Science No. 87.

Aubin, R. (1975). Some generalization heuristics in proofs by induction. In Huet, G. and Kahn, G., (eds.), *Actes du Colloque Construction: Amélioration et vérification de Programmes.* Institut de recherche d'informatique et d'automatique.

Aubin, R. (1976). *Mechanizing Structural Induction.* Unpublished Ph.D. thesis, University of Edinburgh.

Baker, S. (1989). The automatic derivation of constructive proofs from classical proofs. Unpublished M.Sc. thesis, University of Edinburgh.

Baker, S. (1990). Notes on the implementation of the cut elimination theorem. Working Paper 231, Dept. of Artificial Intelligence, Edinburgh.

Baker, S. (1992). CORE manual. Technical Paper 10, Dept. of Artificial Intelligence, Edinburgh.

Baker, S., Ireland, A. and Smaill, A. (1992). On the use of the constructive omega rule within automated deduction. In Voronkov, A., (ed.), *International Conference on Logic Programming and Automated Reasoning - LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 214–225. Springer-Verlag.

Barnett, R., Basin, D. and Hesketh, J. (1992). A recursion planning analysis of inductive completion. Research Paper 518, Dept. of Artificial Intelligence, Edinburgh, Presented to the Second Annual Symposium on Artificial Intelligence and Mathematics.

Basin, David A. (1991). Extracting circuits from constructive proofs. Research Paper 533, Dept. of Artificial Intelligence, Edinburgh, Also appeared in Proceedings of the IFIP-IEEE International Workshop on Formal Methods in VLSI Design, Miami USA, 1991.

Bernays, P. (1983). On platomism in mathematics. In Benacerraf, P. and Putnam, H., (eds.), *Philosophy of Mathematics: selected readings*, pages 258–271. Cambridge University Press. English translation by C. Parsons.

Bishop, E. (1967). *Foundations of Constructive Analysis.* McGraw-Hill, New York.

Biundo, S., Hummel, B., Hutter, D. and Walther, C. (1986). The Karlsruhe induction theorem proving system. In Siekmann, Joerg, (ed.), *8th Conference on Automated Deduction*, pages 672–674. Springer-Verlag. Springer Lecture Notes in Computer Science No. 230.

Boolos, G.S. and Jeffrey, R.C. (1980). *Computability and Logic.* Cambridge University Press.

Boswell, R. (1987). *Extensions of Analytic Concept-Learning applied to Problem-Solving.* Unpublished Ph.D. thesis, University of Edinburgh.

Bowen, K.A. and Kowalski, R.A. (1982). Amalgamating language and metalanguage in logic programming. In Clark, K. *et al*, (eds.), *Logic Programming*, pages 153–172. Academic Press.

Boyer, R.S. and Moore, J.S. (1979). *A Computational Logic.* Academic Press, ACM monograph series.

Boyer, R.S. and Moore, J.S. (1988). *A Computational Logic Handbook.* Academic Press, Boston.

Boyer, R.S. and Moore, J.S. (1990). A theorem prover for a computational logic. In *Proceedings of the Tenth International Conference on Automated Deduction.* Kaiserlauten, Germany.

Brouwer, L.E.J. (1964). Intuitionism and formalism, plus other writings. In Benacerraf, P. and Putnam, H., (eds.), *Philosophy of Mathematics: Selected Readings.* Englewood Cliffs, N.J.

Buchholz, W. and Wainer, S.S. (1987). Provably computable functions and the fast growing hierarchy. In Simpson, S.G., (ed.), *Logic and Combinatorics*, volume 65, pages 179–198. American Mathematical Society.

Bundy, A. and Welham, B. (1981). Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212. Also available from Edinburgh as DAI Research Paper 121.

Bundy, A. (1983). *The Computer Modelling of Mathematical Reasoning.* Academic Press, Second Edition.

Bundy, A., van Harmelen, F., Horn, C. and Smaill, A. (1990). The Oyster-Clam system. In Stickel, M.E., (ed.), *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

Burstall, R.M. and Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67.

Buxton, J.N. and Randell, B. (1969). Software engineering techniques. Report on a conference sponsored by the nato science committee, rome, italy, NATO Science Committee (April 1970).

Carnap, R. (1934). *Logische Syntax der Sprache.* Springer, Translated into English as (Carnap, 1937).

Castaing, J. (1985). How to facilitate the proof of theorems by using the induction-matching, and by generalisation. In Joshi, Aravind, (ed.), *Proceedings of the Ninth IJCAI*, pages 1208–1213. IJCAI.

Chin, W.N. (1990). *Automatic Methods for Program Transformation.* Unpublished Ph.D. thesis, Imperial College.

Chisholm, P. (July 1988). *Investigations into Martin-Lof Type Theory as a Programming Logic.* Unpublished Ph.D. thesis, Department of Computer Science, Heriot-Watt University, Edinburgh.

Constable, R.L. et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., New Jersey.

Constable, R.L., Knoblock, T.B. and Bates, J.L. (1985). Writing programs that construct proofs. *Journal of Automated Reasoning*, pages 285–326.

Constable, R.L., Allen, S.F., Bromley, H.M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

Dalen, Van, (ed.). (1981). *Brouwer's Cambridge Lectures on Intuitionism.* Cambridge University Press.

Dawson, M. (1992). A generic logic environment. In Voronkov, A., (ed.), *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 214–225. Springer-Verlag.

der Waerden, Van. (1971). How the proof of Baudet's conjecture was found. In Mirsky, L., (ed.), *Papers presented to Richard Rado on the occasion of his sixty-fifth birthday*, pages 252–260. Academic Press, London-New York.

Donat, M.R. and Wallen, L.A. (1988). Learning and applying generalised solutions using higher order resolution. In Lusk, E. and Overbeek, R., (eds.), *Lecture Notes in Computer Science*, volume 310, pages 41–60. Springer-Verlag.

Dummett, M. (1975). The philosophical basis of intuitionistic logic. In Rose, H.E. and Shepherdson, J.C., (eds.), *Logic Colloquium '73*, pages 5–40. North Holland.

Dummett, M. (1977). *Elements of Intuitionism.* Oxford Logic Guides. Oxford Univ. Press, Oxford.

Dummett, M. (1991). *The Logical Basis of Metaphysics.* Duckworth, U.S.A.

Fahlman, S.E. (1979). *NETL: A System for Representing and Using Real-World Knowledge.* MIT Press.

Feferman, S. and Spector, C. (1962). Incompleteness along paths in progressions of theories. *Journal of Symbolic Logic*, 27:383–390.

Feferman, S. (1960). Arithmetization of metamathematics in a general setting. *Fundamenta Mathematica*, 49:35–92.

Feferman, S. (1962). Transfinite recursive progressions of axiomatic theories. *Journal of Symbolic Logic*, 27:259–316.

Feferman, S. (1989). Finitary inductively presented logics. In *Logic Colloquium '88*, pages 191–220, Amsterdam. North-Holland.

Feferman, S. (1991). Reflecting on incompleteness. *Journal of Symbolic Logic*, 56:1–49.

Fikes, R.E., and Nilsson, N.J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

Friedman, H. and Scedrov, A. (1985). Arithmetic transfinite induction and recursive well-orderings. *Advances in Mathematics*, 56:283–294.

Gaifman, H. and Shapiro, E. (1988). Fully abstract compositional semantics for logic programs. Technical Report CS88-15, Department of Computer Science, Weizmann Institute of Science.

Gallier, J. (1986). *Logic for Computer Science.* Harper & Row, New York.

Gentzen, G. (1969). *The Collected Papers of Gerhard Gentzen.* North Holland, edited by Szabo, M.E.

Gilmore, P.C. (1970). An examination of the geometry theorem-proving machine. *Artificial Intelligence*, 1:171–87.

Girard, J.-Y. (1987). *Proof Theory and Logical Complexity*, volume 1. Bibliopolis, Naples.

Giunchiglia, F. and Walsh, T. (1989). Abstract Theorem Proving. In *Proceedings of IJCAI 89*, pages pp 372–377. International Joint Conference on Artificial Intelligence. Also available from Edinburgh as DAI Research Paper No 430.

Gödel, K. (1931). Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 38:173–98. English translation in (Heijenoort, 1967).

Gödel, K. (1958). Über enine bisher noch nicht benützte erweiterung des finiten standpunktes. *Dialectica*, 12:280–287. English translation in (Hodges & Watson, 1980).

Goguen, J.A. (1980). How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type implementation. In Bibel, W. and Kowalski, R., (eds.), *5th Conference on Automated Deduction*, volume 87 of *LNCS*, pages 356–373. Springer Verlag.

Goldson, D., Reeves, S. and Bornat, R. (1992). A review of several programs for the teaching of logic. Technical report, Department of Computer Science, Queen Mary and Westfield College, University of London, Submitted to CACM.

Gordon, M. (1988). HOL: A proof generating system for higher-order logic. In Birtwistle, G. and Subrahmanyam, P.A., (eds.), *VLSI Specification, Verification and Synthesis.* Kluwer.

Gordon, M.J., Milner, A.J. and Wadsworth, C.P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag.

Grzegorczyk, A., Motowski, A. and Ryll-Nardzewski, C. (June 1958). The classical and the ω-complete arithmetic. *The Journal of Symbolic Logic*, 23(2):188–206.

Hayes, P. (1977). In defence of logic. In *Proceedings of IJCAI-77*, pages 559–565. International Joint Conference on Artificial Intelligence.

Heijenoort, J van. (1967). *From Frege to Gödel: a source book in Mathematical Logic, 1879-1931.* Harvard University Press, Cambridge, Mass.

Hesketh, J. (1991a). Generalising variables apart using middle out reasoning. Bluebook note 645, Mathematical Reasoning Group, Dept AI, Edinburgh.

Hesketh, J.T. (1991b). *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* Unpublished Ph.D. thesis, University of Edinburgh.

Heyting, A. (1934). *Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie.* Springer, Berlin.

Heyting, A. (1956). *Intuitionism, an Introduction.* North-Holland, Amsterdam.

Hilbert, D. (1930). *Die Grundlebung der elementahren Zahlenlehre*, volume 104. Mathematische Annalen.

Hogger, C.J. (1984). *Introduction to logic programming.* Academic Press.

Horn, C. and Smaill, A. (1990). Theorem proving with Oyster. Research Paper 505, Dept. of Artificial Intelligence, Edinburgh, To appear in Procs IMA Unified Computation Laboratory, Stirling.

Horn, L.R. (1972). *On the semantic Properties of Logical Operators in English.* Unpublished Ph.D. thesis, University of California at Los Angeles.

Howard, W.A. (1980). The formulae-as-types notion of construction. In Seldin, J.P. and Hindley, J.R., (eds.), *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press.

Huet, G. and Hullot, J. (1982). Proofs by induction in equational theories with constructors. *Journal of the Association for Computing Machinery*, 25(2).

Hummel, B. (June 1987). An investigation of formula generalisation heuristics for induction proofs. Interner Bericht 6/87, Universitaet Karlsruhe.

Hutter, D. (1990). Guiding inductive proofs. In Stickel, M.E., (ed.), *10th International Conference on Automated Deduction*, pages 147–161. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449.

Ignjatovic, A. (1988). Hilbert's program and the ω-rule. Technical report, Group in Logic and Methodology of Science, Univ. of California at Berkeley.

Isaacson, D. (1987). Arithmetical truth and hidden higher-order concepts. In *Logic Colloquim '85*, pages 147–169. North-Holland. Edited by Paris Logic Group.

Isaacson, D. (1992). Some considerations on arithmetic truth and the ω-rule. In Detlefsen, Michael, (ed.), *Proof, Logic and Formalisation*. Routledge.

J.-Y. Girard, Y. Lafont and Taylor, P. (1989). *Proofs and Types*. Cambridge University Press.

Kaye, R. (1991). *Models of Peano Arithmetic*, volume 15 of *Oxford Logic Guides*. Clarendon Press, Oxford.

Kedar-Cabelli, S. and McCarty, L.T. (1987). Explanation-based generalization as resolution theorem proving. In Langley, P., (ed.), *Proceedings of the 4th International Machine Learning Workshop*, pages 383–389, University of California, Irvine, CA. Morgan Kaufmann.

Keisler, H.J. (1971). *Model Theory for Infinitary Logic: Logic with Countable Conjunctions and Finite Quantifiers*. North-Holland.

Kinber, E.B. and Brazma, A.N. (1990). Models of inductive synthesis. *Journal of Logic Programming*, 9:221–233.

Kleene, S. (1962). *Introduction to Metamathematics*. North Holland, Amsterdam.

Kreisel, G. and Lévy, A. (1968). Reflection principles and their use for establishig the complexity of axiomatic systems. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 14:97–142.

Kreisel, G. (1962a). Five notes on transfinite progressions. Technical Report 5, Applied Math. and Statistics Lab., Stanford.

Kreisel, G. (1962b). Foundations of intuitionistic logic. In Nagel, Suppes and Tarski, (eds.), *Logic, Methodology and Philosophy of Science*, pages 198–210. Stanford University Press.

Kreisel, G. (1965). Mathematical logic. In Saaty, T.L., (ed.), *Lectures on Modern Mathematics*, volume III, pages 95–195. John Wiley and Sons.

Kreisel, G. (1971). A survey of proof theory ii. In Fenstad, J.E., (ed.), *Studies in Logic and the Foundations of Mathematics: Proceedings of the Second Scandinavian Logic Symposium*, volume 63, pages 109–170. North Holland.

Lakatos, I. (1976). *Proofs and refutations: The logic of Mathematical discovery*. Cambridge University Press.

Löpez-Escobar, E.G.K. (1976). On an extremely restricted ω-rule. *Fundamenta Mathematicae*, 90:159–72.

Löpez-Escobar, E.G.K. (1977). Infinite rules in finite systems. In Arruda, Da Costa and Chuaqui, (eds.), *Non-classical Logics, Model Theory and Computability.* North Holland. Studies in Logic 89.

Lucas, J.R. (1970). *The Freedom of the Will.* Oxford Clarendon Press.

Luo, Z. and Pollack, R. (May 1992). Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh.

Madden, P. (1989). The specialization and transformation of constructive existence proofs. In Sridharan, N.S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Morgan Kaufmann.

Martin-Lof, P. (1970). *Notes on Constructive Mathematics.* Almqvist and Wiksell, Stockholm.

Matthews, S. (July 1989). Seldon manual. Technical report, Dept Artificial Intelligence, University of Edinburgh.

Matthews, S., Smaill, A and Basin, D. (1991). Experience with FS_0 as a framework theory. In *Proceedings of the Second Workshop on Logical Frameworks*, Edinburgh, Scotland.

Mendelson, E. (1964). *Introduction to Mathematical Logic.* van Nostrand Reinhold Co.

Michalski, R.S. (1983). A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161.

Minsky, M. (1963). Steps towards artificial intelligence. In Feigenbaum, E.A. and Feldman, J., (eds.), *Computers and Thought*, pages 406–450. McGraw-Hill.

Mitchell, T.M. (1978). *Version Spaces: An approach to concept learning.* Unpublished Ph.D. thesis, Stanford University.

Mitchell, T.M. (1982a). Generalization as search. *Artificial Intelligence*, 18:203–226.

Mitchell, T.M. (1982b). Toward combining empirical and analytical methods for inferring heuristics. Technical Report LCSR-TR-27, Laboratory for Computer Science Research, Rutgers University.

Mitchell, T.M., Utgoff, P. E., Nudel, B. and Banerji, R. (1981). Learning problem-solving heuristics through practice. In *Proceedings of IJCAI-81*, pages 127–134. International Joint Conference on Artificial Intelligence.

Mitchell, T.M., Utgoff, P. E. and Banerji, R. (1983). Learning by experimentation: Acquiring and modifying problem-solving heuristics. In Michalski, R.S., Carbonell, J.F. and Mitchell, T.M., (eds.), *Machine Learning*, pages 163–190. Tioga Press.

Mitchell, T.M., Keller, R.M. and Kedar-Cabelli, S.T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80. Also available as Tech. Report ML-TR-2, SUNJ Rutgers, 1985.

Muggleton, S. (1988). A strategy for constructing new predicates in first-order logic. In *Proceedings of the Third European Working Session on Learning*. Pitman Publishing.

Nelson, G.C. (1971). A further restricted ω-rule. *Colloquium Mathematicum*, 23.

O'Rorke, P.V. (January 1987). *Explanation-Based Learning Via Constraint Posting and Propagation*. Unpublished Ph.D. thesis, Department of Computer Science, University of Illinois.

Parsons, C. (1958). Elimination of a restricted infinite induction. *J. Sym. Logic*, 23:106. Abstract of paper given at 22nd annual meeting of the Association for Symbolic Logic.

Paulson, L. (1986). Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–258.

Penrose, R. (1989). *The Emperor's New Mind*. Vintage.

Plaisted, D.A. (1980). Abstraction mappings in mechanical theorem proving. In *Proceedings of the 5th Conference on Automated Deduction*.

Plotkin, G. (1969). A note on inductive generalization. In Michie, D and Meltzer, B, (eds.), *Machine Intelligence 5*, pages 153–164. Edinburgh University Press.

Plummer, D. and Bundy, A. (February 1984). Gazing: Identifying potentially useful inferences. Working Paper 160, Dept. of Artificial Intelligence, Edinburgh.

Prawitz, D. (1965). *Natural deduction: a proof-theoretical study*. Almqvist & Wiksell, Stockholm.

Prawitz, D. (1971). Ideas and results in proof theory. In Fenstad, J.E., (ed.), *Studies in Logic and the Foundations of Mathematics: Proceedings of the Second Scandinavian Logic Symposium*, volume 63, pages 235–307. North Holland.

Prawitz, D. (1977). Meaning and proofs: on the conflict between classical and intuitionistic logic. *Theoria*, 43:2–40.

Reddy, U. S. (1990). Term rewriting induction. In *Proc. of Tenth International Conference on Automated Deduction*. Springer-Verlag.

Robinson, J.A. (1965). A machine oriented logic based on the resolution principle. *J Assoc. Comput. Mach.*, 12:23–41.

Rosser, B. (September 1937). Gödel-theorems for non-constructive logics. *JSL*, 2(3):129–137.

Rouveirol, C. (August 1990). Saturation: Postponing choices when inverting resolution. In *Proceedings of ECAI-90*, pages 557–562, Stockholm.

Schütte, K. (1960). *Beweistheorie.* Springer-Verlag.

Schütte, K. (1977). *Proof Theory.* Springer-Verlag.

Schwichtenberg, H. (1977). Proof theory: Some applications of cut-elimination. In Barwise, (ed.), *Handbook of Mathematical Logic*, pages 867–896. North-Holland.

Shavlik, J.W. and Jong, G.F. De. (March 1989). Learning in mathematically based domains: Understanding and generalising obstacle cancellations. *Artificial Intelligence.*

Shoenfield, J.R. (1959). On a restricted ω-rule. *Bull. Acad. Sc. Polon. Sci., Ser. des sc. math., astr. et phys.*, 7:405–7.

Skolem, T. (1934). Über die nichtcharakterisierbarkeit der zahlenreihe mittels endlich oder abzählbar unendlich vieler aussagen mit ausschliesslich zahlenvariablen. *Fund. Math.*, 23.

Sundholm, B.G., (1978). The omega rule: A survey. University of Oxford, B. Phil Thesis.

Sundholm, B.G. (1983a). Constructions, proofs, and the meaning of the logical constants. *Journal of Philosophical Logic*, 12:151–172.

Sundholm, B.G. (1983b). *A Survey of the Omega Rule.* Unpublished Ph.D. thesis, University of Oxford.

Sundholm, B.G. (1986). Proof theory and meaning. In Gabbay, D. and Guenthner, F., (eds.), *Handbook of Philosophical Logic*, volume 3, pages 471–506. D. Reidel Publishing Company.

Takahashi, M. (1970). A theorem on the second order arithmetic with the ω-rule. *Journal of the Mathematical Society of Japan*, 22.

Takeuti, G. (1987). *Proof theory.* North-Holland, 2 edition.

Tarski, A. (1933). Einige betrachtungen über die begriffe der ω-widerspruchsfreiheit und der ω-vollständigkeit. *Monatshefte für Mathematik und Physik*, 40:97–112. English translation by J.H. Woodger in (Tarski, 1956).

Tarski, A. (1936). Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405. English translation in (Tarski, 1956).

Thomas, M. and Jantke, K.P. (1989). Inductive inference for solving divergance in Knuth-Bendix. In Jantke, K.P., (ed.), *Analogical and Inductive Inference, Procs. of AII'89*, pages 288–303. Springer-Verlag.

Thomas, M. and Watson, P. (May 1991). Solving divergence in Knuth-Bendix completion by enriching signatures. Technical report, Department of Computing Science, University of Glasgow.

Troelestra, A.S. (1982). Metamathematical investigation of intuitionistic arithmetic and analysis. In Dold, A. and Eckmann, B., (eds.), *Lecture Notes in Mathematics*, volume 344. Springer-Verlag.

Tucker, J.V., Wainer, S.S. and Zucker, J.I. (1990). Provable computable functions on abstract-data-types. *Lecture Notes in Computer Science*, 443:660–673.

Turing, A.M. (1939). Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, 45:161–228.

Twidale, M. (1989). Intermediate representations for student error diagnosis and support. In Bierman, D., Breuker, J. and Sandberg, J., (eds.), *Artificial Intelligence: Synthesis and Reflection*, pages 298–306, Amsterdam. IOS.

van Harmelen, F. and Bundy, A. (1987). Explanation-based generalization = partial evaluation. Research Paper 347, Dept. of Artificial Intelligence, Edinburgh, Published in the AI Journal, vol. 36, no. 3.

van Harmelen, F. (1989). The clam proof planner. Dai technical paper no. 4, Dept Artificial Intelligence, University of Edinburgh.

Walther, C. (1988). Argument-bounded algorithms as a basis for automated termination proofs. In Lusk, R. and Overbeek, R., (eds.), *9th Conference on Automated Deduction*, pages 602–621. Springer-Verlag. Revised version to appear in AI Journal.

Walther, C. (1992). Mechanizing mathematical induction. In B.M. Gabbay, C.J. Hogger and Robinson, J.A., (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, Oxford. in press.

Weyhrauch, R.W. (1980). Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170.

Whitehead, A.N. and Russell, B. (1925). *Principia Mathematica*, volume 1. CUP, 2nd edition.

Wickstead, P. and Comford, F.M. [Trans.]. (1929). *Aristotle: The Physics, Book III*. W. Heinemann Ltd.

Winston, P. (1975). Learning structural descriptions from examples. In Winston, P.H., (ed.), *The psychology of computer vision*. McGraw Hill.

Yoccoz, S. (1989a). Constructive aspects of the omega-rule: Application to proof systems in computer science and algorithmic logic. *Lecture Notes in Computer Science*, 379:553–565.

Yoccoz, S. (1989b). *Recursive ω-rule for proof systems*, volume 31, pages 291–294. North-Holland.

Zermelo, E. (1932). Über die stufen der quantifikation und die logik der unedlichen. *Jahresbericht der Deutschen Mathematiker Verieningung*, 41:85–88.

Zucker, J. (1974). The correspondence between cut-elimination and normalisation. *Annals of Mathematical Logic*, 7:1–112.

Appendix A

Sequent Calculus

The following provides a brief introduction to Gentzen's Sequent Calculus (Gentzen, 1969), and to the syntax of Seldon[1]. This appendix has been included in order to present a grounding in the logic used in the overall implementation, and because it is necessary to be familiar with the sequent calculus in order to understand the ideas involved in the thesis.

A.1 Syntax

The calculus consists of the normal syntax for predicate calculus, namely variables, constant symbols, functions and well-formed formulae built up from atomic propositions, the connectives $\vee, \wedge, \rightarrow, \neg$ and the quantifiers $\forall, \exists$. See Section 4.1 for further details.

A.2 Sequents

A sequent is written in Gentzen's system as

$$F_1, \ldots, F_n \vdash G_1, \ldots, G_m$$

where F_i, G_j are well-formed formulae. This corresponds to

$$\{F_1 \wedge \ldots \wedge F_n\} \rightarrow \{G_1 \vee \ldots \vee G_m\}.$$

If, as Gentzen did, finite lists of formulae are used (as opposed to the alternative of using sets) then structural rules are needed to permute and contract formulae in the finite lists.[2] In the case of intuitionistic sequent calculus (and the Seldon

---

[1]Seldon is the system in which implementation is carried out; for further details see Section 10.2.

[2]However, the contraction rule $\frac{A,A,\Gamma \vdash C}{A,\Gamma \vdash C}$ raises the question of assumption classes: see (Zucker, 1974) for further details.

system) multiple conclusions are not allowed. In either case, any sequent having the same formula on both sides of the entailment operator, $\vdash$, is always true. That is to say that $\Gamma, A \vdash A$ is called an axiom as it is obviously not falsifiable. A sequent $\Gamma \vdash \Delta$ is falsifiable if one can made all the members of Γ true and all the members of Δ false.

The left-hand side is referred to as the antecedent, and the right-hand side as the succedent — they correspond to assumptions and conclusions respectively. If the antecedent is empty, that corresponds to the succedent being true, and if the succedent is empty, to the antecedent being false. (However, if both antecedent and succedent are empty, the sequent is false).

The classical representation is in fact equivalent to the intuitionistic representation,[3] together with the law of excluded middle ($\vdash A \vee \neg A$). This fact is exploited within Seldon to allow representation of both classical and intuitionistic theories of arithmetic.[4]

A.3 Prooftrees

Deduction trees are used for investigating proofs. At the bottom is the sequent to be proved, and the tree is developed upwards using the inference rules given in Table A–1, together with structural rules which allow re-ordering or duplication of formulae. The rules given are those for intuitionistic logic, and are obtained from Gentzen's classical sequent rules by restriction to a single consequent.

In each case, the sequent above the line is the premise and that below the line is the conclusion. Since these rules (with the exception of the cut rule) reduce the number of connectives when applied backwards, each branch of the proof will terminate eventually. Hence, a deduction tree is either a sequent which cannot be expanded further by any inference rule, or a tree grown from a sequent by the rules of inference to leaves which cannot be expanded further. It is a prooftree if all its leaves are axioms.

An example of a prooftree is given below — it demonstrates the validity of $P, P \to Q \vdash Q$.

$$\dfrac{\dfrac{}{g : P \vdash P}\,axiom \qquad \dfrac{}{h : Q,\ g : P \vdash Q}\,axiom}{g : P,\ f : P \to Q \vdash Q}\,limp(f)$$

[3] Γ varies over lists of formulae.

[4] See Section 10.2.

Rule	*Rulename*	*Rule*	*Rulename*
$\frac{\Gamma\vdash C}{\Gamma,f{:}A\vdash C}$	lthin(f)	$\frac{\Gamma\vdash\bot}{\Gamma\vdash G}$	rthin
$\frac{\Gamma\vdash A}{f{:}\neg A,\Gamma\vdash}$	lnot(f)	$\frac{A,\Gamma\vdash}{\Gamma\vdash\neg A}$	rnot
$\frac{A,B,\Gamma\vdash C}{f{:}A\wedge B,\Gamma\vdash C}$	land(f)	$\frac{\Gamma\vdash A\quad\Gamma\vdash B}{\Gamma\vdash A\wedge B}$	rand
$\frac{A,\Gamma\vdash C\quad B,\Gamma\vdash C}{f{:}A\vee B,\Gamma\vdash C}$	lor(f)	$\frac{\Gamma\vdash A}{\Gamma\vdash A\vee B}$	ror(left)
$\frac{\Gamma\vdash A\quad\Gamma,A\vdash C}{\Gamma\vdash C}$	cut	$\frac{\Gamma\vdash B}{\Gamma\vdash A\vee B}$	ror(right)
$\frac{\Gamma\vdash A\quad B,\Gamma\vdash C}{f{:}A\to B,\Gamma\vdash C}$	limp(f)	$\frac{A,\Gamma\vdash B}{\Gamma\vdash A\to B}$	rimp
$\frac{\Phi(t),\Gamma\vdash C}{f{:}\forall x\Phi(x),\Gamma\vdash C}$ t an arbitrary term	luni(f,t)	$\frac{\Gamma\vdash\Phi(a)}{\Gamma\vdash\forall x\Phi(x)}$ a not in lower sequent	runi(a)
$\frac{\Phi(a),\Gamma\vdash C}{f{:}\exists x\Phi(x),\Gamma\vdash C}$ a not in lower sequent	lexi(f,a)	$\frac{\Gamma\vdash\Phi(t)}{\Gamma\vdash\exists x\Phi(x)}$ t an arbitrary term	rexi(t)

Table A–1: The Rules of Sequent Calculus in the Form Used by Seldon

Appendix B

Useful Commands in Seldon/Oyster

B.1 Start-up Commands/Commands for Traversing Tree

oyster to start Oyster

seldon to start Seldon

demo to start CORE

up go up one node in the tree

down go down one node in the tree

down(N) go to the Nth subgoal

top go to the top of the prooftree

next go to the next open subgoal

B.2 Commands for Manipulating Proofs/Writing Tactics

load_thm(referencename, savedname) to load a saved theorem

create_thm(referencename, user) plus typing sequent after colon given to load a new theorem

select(referencename) selects the particular theorem one wishes to work with

save_thm(referencename, savedname) saves the current state of a proof

snapshot(savedname) produces a readable version of the prooftree below the current node

status(S) instantiates S to the status of the current goal: partial, complete or incomplete

goal(G) instantiates G to the current goal

hypothesis(H) instantiates H to a hypothesis: this may be used to generate hypotheses one by one, or to find one that matches some pattern

hyp_list(X) instantiates X to the list of hypotheses of the current node

B.3 The Specification of Positions of Subterms Within a Formula

It is necessary to tackle the problem of specification of positions of subterms within a formula in order to be able to store the exact subexpression to which a rewrite rule is to be applied. Fortunately, there is a provision for this within the CLAM system, namely by the use of the predicate "exp_at". Information about the position of a sub-formula is included in the general proof representation, using this predicate.

The latter is of the form 'exp_at(+Exp,?Pos,?SubExp)'. The expression Exp contains a subexpression SubExp at position Pos. Positions are lists of integers representing tree-coordinates in the syntax-tree of the expression, but in reverse order. Coordinate 0 represents the function symbol of an expression. One may find the SubExp at a given Pos, or alternatively find the Pos of a given SubExp.

The following Prolog transcript illustrates use of the predicate.

```
| ?- exp_at(f(g(2,x),3),X,Y).

X = [], Y = f(g(2,x),3) ;
X = [2], Y = 3 ;
X = [1], Y = g(2,x) ;
X = [0], Y = f ;
X = [2,1], Y = x ;
X = [1,1], Y = 2 ;
X = [0,1], Y = g ;
no
```

Appendix C

Comparison of List Proofs: Summary

This appendix summarises and compares the main alternative approaches towards the proof of various list examples which involve accumulators.

C.1 $\forall l\ len(rev(l)) = len(l)$

Proof by induction:
Requires $rev(T)$ to be generalised to U, as shown below. The following is NQTHM's method:
$len(rev(H :: T)) = len(H :: T)$
$len(rev(T) <> (H :: nil)) = len(H :: T) = s(len(T)) = s(len(rev(T)))$ *by hyp*
Note that the general proof carries on from this stage, working on the structure of T.
Generalise to: $len(U <> (H :: nil)) = s(len(U))$.
Use induction on U. $len((V :: U) <> (H :: nil)) = s(len(V :: U))$
$len(V :: (U <> (H :: nil))) = s(s(len(U)))$
$s(len(U <> (H :: nil))) = s(s(len(U)))$
$s(s(len(U))) = s(s(len(U)))$ *by hyp*
(plus base cases)

First Generalisation:
Generalisation of this example has as far as I know not been carried out by anyone else's generalisation method, although the generalisation provided by J. Hesketh's method, as given in (Hesketh, 1991b), if obtained, would be $\forall al\ len(rev(l) <> a) = len(a <> l)$, which may be proved using two inductions, plus C.1, where the latter is:

$$(X <> Y) <> Z = X <> (Y <> Z) \qquad \text{(C.1)}$$

(cf. Section 8.6 for a discussion of the relevance of the associativity of append in these proofs). The induction is as follows:
$\forall a'\ len(rev(V) <> a') = len(a' <> V) \vdash len(rev(U :: V) <> a) = len(a <> U :: V)$
ie.... $\vdash len((rev(V) <> (U :: nil)) <> a) = len(a <> (U :: V))$
ie.... $\vdash len((rev(V)) <> ((U :: nil) <> a)) = len(a <> (U :: V))$ *by C.1*

$\ldots \vdash len(((U :: nil) <> a) <> V) = \ldots$ *by hyp*
$\ldots \vdash len(U :: a <> V) = len(a <> (U :: V))$
$\ldots \vdash s(len(a <> V)) = len(a <> (U :: V))$
Induction on a (plus base cases) gives:
$s(len(b :: a <> V)) = len(b :: a <> (U :: V))$
$s(s(len(a <> V))) = s(len(a <> (U :: V))) = s(s(len(a <> V)))$ *by hyp*

Second Generalisation:
This generalisation is obtained using the 'guiding method', as shown above, and is $\forall al\ len(rev(l) <> a) = len(rev(a) <> l)$. It may be proved using only one induction, plus C.1. (Note that this the the generalisation suggested by the general proof).

$\forall a'\ len(rev(V) <> a') = len(rev(a') <> V) \vdash len(rev(U :: V) <> a) = len(rev(a) <> U :: V)$
ie.... $\vdash len(rev(V) <> (U :: nil)) <> a) = len(rev(a) <> (U :: V))$
ie.... $\vdash len(rev(V) <> ((U :: nil) <> a)) = len(rev(a) <> (U :: V))$ *(by C.1)*
ie.... $\vdash len(rev((U :: nil) <> a) <> rev(V)) = \ldots$ *(by HYP)*
ie.... $\vdash len(rev((U :: a) <> V)) = len(rev(a) <> (U :: V))$
ie.... $\vdash len((rev(a) <> (U :: nil)) <> V) = \ldots$
ie.... $\vdash len(rev(a) <> ((U :: nil) <> V)) = len(rev(a) <> (U :: V))$ *(by C.1)*
ie.... $\vdash len(rev(a) <> U :: V) = len(rev(a) <> U :: V)$
So, this is a better generalisation since it requires only one induction.

General Proof of Original Goal:
$P(s(n))$ does not reduce to $P(n)$, but the general proof carries through. Compare the general proof with the (failed) proof by induction.

$P(s(n)) \equiv len(rev(\mathcal{L}(s(n)))) = len(\mathcal{L}(s(n)))$ *(part of GP)*
$\equiv len(rev(x_{s^n(0)} :: \mathcal{L}(n))) = len(x_{s^n(0)} :: \mathcal{L}(n))$
$\equiv len((rev(\mathcal{L}(n)) <> (x_{s^n(0)} :: nil)) = len((x_{s^n(0)} :: nil) <> \mathcal{L}(n))$
which is not equivalent to $P(n)$.

General Proof of Generalised Goal:
$Q(s(n)) \equiv len(rev(\mathcal{L}(s(n))) <> a) = len(rev(a) <> (\mathcal{L}(s(n))))$
$\equiv len(rev(x_{s^n(0)} :: \mathcal{L}(n)), .a) = len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n)))$
$\equiv len((rev(\mathcal{L}(n)) <> (x_{s^n(0)} :: nil)) <> a) = len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n)))$
$\equiv len(rev(\mathcal{L}(n)) <> ((x_{s^n(0)} :: nil) <> a) = len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n)))$ *(by C.1)*
$\equiv len(rev(\mathcal{L}(n)) <> (x_{s^n(0)} :: a)) = len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n)))$
$\equiv Q(n)$
if $len(rev(x_{s^n(0)} :: a) <> \mathcal{L}(n)) = len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n)))$
ie. if $len((rev(a) <> (x_{s^n(0)} :: nil)) <> \mathcal{L}(n)) = len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n)))$
ie. if $len(rev(a) <> ((x_{s^n(0)} :: nil) <> \mathcal{L}(n)))$ *(by C.1)* $= len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n)))$
ie. if $len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n))) = len(rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n)))$ *(OK)*
$Q(s(n))$ does reduce to $Q(n)$ (but one must force this reduction, in the sense that it is necessary to match the right hand side of the equation with the desired form, ie. to check that $rev(x_{s^n(0)} :: a) <> \mathcal{L}(n) =^? rev(a) <> (x_{s^n(0)} :: \mathcal{L}(n))$. Thus, we

may have a linear general proof of the form

$$\forall b\ len(rev(\mathcal{L}(n)) <> b) = len(rev(b) <> \mathcal{L}(n)) \equiv Q(n)$$

$$\vdots$$

$$\forall a\ len(rev(\mathcal{L}(s(n))) <> a) = len(rev(a) <> \mathcal{L}(s(n))) \equiv Q(s(n))$$

In fact, $b = x_{s^n(0)} :: a$. Since there is a linear form, induction will work on Q, which is the case. This process of linearisation may be thought of as the search for an inductive proof.

Note that in this example, the proof by induction proceeds when blocked by generalisation of $rev(T)$ to U, and then induction on U. This sort of approach is not available for many examples. The general proof proceeds by breaking down the structure of T, by means of the representation of members of a type, which allows additional rewrite rules to be applied.

C.2 $\forall l\ rotate(len(l), l) = l$

Proof by induction:
This is blocked.

$$\begin{aligned} rotate(len(T), T) = T &\vdash rotate(len(H :: T), H :: T) = H :: T \\ ie.\ \ldots &\vdash rotate(s(len(T)), H :: T) = H :: T \\ ie.\ \ldots &\vdash rotate(len(T), T <> (H :: nil)) = H :: T \\ & \quad BLOCKED \end{aligned}$$

Note that the general proof allows continuation of this proof, by the structure of T being broken down.

Generalisation:
This is $\forall al\ rotate(len(l), l <> a) = a <> l$, which is provable using one induction, plus C.1. (But it is necessary to match $x_{s^n(0)} :: \mathcal{L}(n) =^? (x_{s^n(0)} :: nil) <> \mathcal{L}(n)$ to suggest this generalisation from the general proof).

$\forall a' rotate(len(T), T <> a') = a' <> T \vdash rotate(len(H :: T), H :: T <> a) = a <> H :: T$
$ie.\ldots \vdash rotate(s(len(T)), H :: (T <> a)) = a <> H :: T$
$ie.\ldots \vdash rotate(len(T), (T <> a) <> (H :: nil)) = a <> (H :: T)$
$ie.\ldots \vdash rotate(len(T), T <> (a <> (H :: nil))) = a <> H :: T$ $(by\ C.1)$
$ie.\ldots \vdash (a <> (H :: nil)) <> T = a <> H :: T$ $(by\ HYP)$
$ie.\ldots \vdash a <> ((H :: nil) <> T) = a <> H :: T$ $(by\ C.1)$
$ie.\ldots \vdash a <> H :: T = a <> H :: T$
(Plus base case).

General Proof of Original Goal:
$P(s(n))$ does not reduce to $P(n)$.

$P(s(n)) \equiv rotate(len(\mathcal{L}(s(n))), \mathcal{L}(s(n))) = \mathcal{L}(s(n))$ (*1st line of GP*
$\equiv rotate(s^n(0), x_{s^n(0)} :: \mathcal{L}(n)) = \mathcal{L}(s(n))$
$\equiv rotate(len(\mathcal{L}(n)), \mathcal{L}(n) <> (x_{s^n(0)} :: nil)) = x_{s^n(0)} :: \mathcal{L}(n)$
which is not equivalent to $P(n)$.

General Proof of Generalised Goal:
$Q(s(n))$ does reduce to $Q(n)$, but it is necessary to match goals, and use C.1 in the proof.

$Q(s(n)) \equiv rotate(len(\mathcal{L}(s(n))), \mathcal{L}(s(n)) <> a) = a <> \mathcal{L}(s(n))$
$\equiv rotate(len(\mathcal{L}(s(n))), (x_{s^n(0)} :: \mathcal{L}(n)) <> a) = a <> (x_{s^n(0)} :: \mathcal{L}(n))$
$\equiv rotate(len(\mathcal{L}(s(n))), x_{s^n(0)} :: (\mathcal{L}(n) <> a)) = a <> (x_{s^n(0)} :: \mathcal{L}(n))$
$\equiv rotate(len(\mathcal{L}(s(n))), \mathcal{L}(n) <> (a <> x_{s^n(0)} :: nil)) = a <> (x_{s^n(0)} :: \mathcal{L}(n))$
$\equiv Q(n)$
$if\ (a <> (x_{s^n(0)} :: nil)) <> \mathcal{L}(n) \equiv a <> (x_{s^n(0)} :: \mathcal{L}(n))$
$if\ a <> ((x_{s^n(0)} :: nil)) <> \mathcal{L}(n)) \equiv a <> (x_{s^n(0)} :: \mathcal{L}(n))$
$if\ a <> x_{s^n(0)} :: (nil <> \mathcal{L}(n)) \equiv a <> (x_{s^n(0)} :: \mathcal{L}(n))$
$if\ a <> x_{s^n(0)} :: \mathcal{L}(n) \equiv a <> (x_{s^n(0)} :: \mathcal{L}(n))\ (OK)$

C.3 $\forall l\ rev2(l, nil) = rev(l)$

Proof by induction:
This is blocked.

$rev2(T, nil) = rev(T) \vdash rev2(H :: T, nil) = rev(H :: T)$
$\ldots \vdash rev2(T, H :: nil) = rev(T) <> (H :: nil)$
$\ldots \vdash \ldots = rev2(T, nil) <> (H :: nil)$

The general proof allows continuation, by the structure of T being broken down (thus allowing more rewrite rules to be applied).

Generalisation:
Namely $\forall al\ rev2(l, a) = rev(l) <> a$, which is provable using one induction, plus C.1.

$\forall a'\ rev2(T, a') = rev(T) <> a' \vdash rev2(H :: T, a) = rev(H :: T) <> a$
$\ldots \vdash rev2(T, H :: a) = (rev(T) <> (H :: nil)) <> a$
$\ldots \vdash rev(T) <> (H :: a) = rev(T) <> ((H :: nil) <> a)$ (*by C.1*)
$\ldots \vdash rev(T) <> (H :: a) = rev(T) <> (H :: a)$
(Plus base cases).

In each case, the generalisation proves the original theorem by setting $a = nil$.

General Proof of Original Goal:
$P(s(n))$ does not reduce to $P(n)$.

$P(s(n)) \equiv rev2(\mathcal{L}(s(n)), nil) = rev(\mathcal{L}(s(n)))$(*1st line of GP*)
$\equiv rev2(x_{s^n(0)} :: \mathcal{L}(n), nil) = rev(x_{s^n(0)} :: \mathcal{L}(n))$
$\equiv rev2(\mathcal{L}(n), x_{s^n(0)} :: nil) = rev(\mathcal{L}(n)) <> (x_{s^n(0)} :: nil)$

No more rewrite rules applicable, and this is not equivalent to $P(n)$. This is in fact exactly the same as why the induction proof gets blocked.

General Proof of Generalised Goal:
$Q(s(n))$ does reduce to $Q(n)$.

$Q(s(n)) \equiv rev2(\mathcal{L}(s(n)), a) = rev(\mathcal{L}(s(n))) <> a$
$\equiv rev2(x_{s^n(0)} :: \mathcal{L}(n), a) = rev(x_{s^n(0)} :: \mathcal{L}(n)) <> a$
$\equiv rev2(\mathcal{L}(n), x_{s^n(0)} :: a) = (rev(\mathcal{L}(n)) <> (x_{s^n(0)} :: nil)) <> a$
$(by\ C.1) \equiv rev2(\mathcal{L}(n), x_{s^n(0)} :: a) = rev(\mathcal{L}(n)) <> ((x_{s^n(0)} :: nil) <> a)$
$\equiv rev2(\mathcal{L}(n), x_{s^n(0)} :: a) = rev(\mathcal{L}(n)) <> x_{s^n(0)} :: a$
$\equiv Q(n)(b \equiv x_{s^n(0)} :: a)$

Appendix D

NQTHM Transcript

The following is a transcript returned by the theorem prover NQTHM when asked to prove $\forall x\ x \neq 0 \rightarrow p(x) + s(s(x)) = s(x) + x$. As seen below, the proof fails. Other methods of proof for this example are discussed in Subsections 8.1.5 and 11.4.1.

```
>(prove-lemma 13 ()
(implies (and (natp x) (not (zilchp x)))
                 (equal (esplus (pred x) (suc (suc x)))
                        (esplus (suc x) x)
                 )
        )
)
```

```
This formula can be simplified, using the abbreviations ZILCHP, NOT, AND, and
IMPLIES, to:

      (IMPLIES (AND (NATP X) (NOT (EQUAL X (ZILCH))))
               (EQUAL (ESPLUS (PRED X) (SUC (SUC X)))
                      (ESPLUS (SUC X) X))),

which simplifies, rewriting with PRED-SUC, and expanding the definitions of
ZILCHP and ESPLUS, to:

      (IMPLIES (AND (NATP X) (NOT (EQUAL X (ZILCH))))
               (EQUAL (ESPLUS (PRED X) (SUC (SUC X)))
                      (SUC (SUC (ESPLUS (PRED X) X))))).

Applying the lemma PRED-ELIM, replace X by (SUC Z) to eliminate (PRED X).  We
use the type restriction lemma noted when PRED was introduced to restrict the
new variable.  This produces:

      (IMPLIES (AND (NATP Z)
                    (NOT (EQUAL (SUC Z) (ZILCH))))
               (EQUAL (ESPLUS Z (SUC (SUC (SUC Z))))
                      (SUC (SUC (ESPLUS Z (SUC Z)))))),

which further simplifies, clearly, to:
```

```
(IMPLIES (NATP Z)
         (EQUAL (ESPLUS Z (SUC (SUC (SUC Z))))
                (SUC (SUC (ESPLUS Z (SUC Z))))),
```

which we would normally push and work on later by induction. But if we must use induction to prove the input conjecture, we prefer to induct on the original formulation of the problem. Thus we will disregard all that we have previously done, give the name *1 to the original input, and work on it.

So now let us return to:

```
(IMPLIES (AND (NATP X) (NOT (ZILCHP X)))
         (EQUAL (ESPLUS (PRED X) (SUC (SUC X)))
                (ESPLUS (SUC X) X))),
```

named *1. Since there is nothing to induct upon, the proof has

********** F A I L E D **********

Appendix E

The CORE System

This appendix extends the description given in Chapter 10 of the Constructive Omega Rule Environment (CORE), which is a system in which ω-proofs may be displayed and investigated. The latter consists in implementation of the system of arithmetic with the ω-rule (cf. $PA_{c\omega}$), which includes generation of individual proofs, the formation of general proofs, demonstration of correctness of the general proof, and the application of rewrite rules. Thus, the implementation allows both the automatic or incremental construction of ω-proofs, and the validations of descriptions of ω-proofs. In addition, it encompasses implementation of the generalisation method proposed in Chapter 8 (ie. the provision of a cut formula via explanation-based generalisation methods). The implementation is carried out within the framework of an interactive theorem-prover with Prolog as the tactic language, namely Seldon, a version of the OYSTER theorem-prover in which the object-level logic is replaced by classical and constructive theories of arithmetic. Relevant code is given in (Baker, 1992).

E.1 Description

This appendix is a technical description of the CORE system, a proof development environment in which a (constructive) version of the ω-rule may be used as a rule of inference. The ω-rule infers the universal application of a proposition from an infinite number of individual cases of that proposition. The full description of the development of the system, together with discussion of general proofs and related theory, has already been given in the main body of the thesis.

In general, cut elimination holds for arithmetical systems with the ω-rule, but not for systems with ordinary induction. Hence in the latter, there is the problem of generalisation, since arbitrary formulae can be cut in. This makes automatic theorem-proving very difficult. An important technique for investigating derivability in formal systems of arithmetic has been to embed such systems into semi-formal systems with the ω-rule. This appendix describes the implementation of such a system.

The implementation is carried out within the framework of an interactive theorem-prover with Prolog as the tactic language, in which the object-level logic is replaced by classical and constructive theories of arithmetic. Any finitely large number of individual instances of proofs of a proposition may be generated automatically by the use of various tactics. The general representation of the proofs is provided by having an initial generalisation from a 'starting' proof, and then by updating this general proof using the other individual proofs, until the general proof seems to have reached a stable form. This general proof is then checked to see if it is indeed the correct one, as described above. The whole of this stage has been automated. There are two options which are allowable from the Seldon proof (in PA or HA) with goal $\Gamma \vdash \forall x P(x)$. One is to ask to use the constructive ω-rule, whereby the system will check to see whether there is a correct general proof, and then return to the former system and close the branch, or else state that this rule may not be used. The user may then continue to investigate other positions in the prooftree. The other option is to ask for an appropriate cut to be carried out in PA (the cut being worked out by the system from the general proof), with a further option to complete the tree as far as possible (using standard theorem-proving techniques). The general proof may be provided automatically, but there is an option in each case to switch temporarily to another system which will allow for the description, manipulation and display of the general proof. The user may specify the proof incrementally, in terms of applications in positions in the tree, plus induction over a distinguished parameter, or all at once — and this is checked. The system builds up a recursive function description of the general proof, and is able to display individual proofs in addition to the general case. A cut formula is automatically suggested from general proofs using an implementation based on the method of explanation-based generalisation, which is a technique for formulating general concepts on the basis of specific training examples, first described in (Mitchell, 1982b).

The following sections provide details of

- the systems allowing development of proofs within arithmetic upon which the implementation is based;
- representation of the ω-rule and its subgoals;
- manipulation of individual proofs;
- provision of a general proof;
- correctness checking;
- generalisation;
- and finally, the interactive system, and how to use it.

Name	*Sequent Rule*	*System-Name*
thin-L	$\frac{\Gamma \vdash G \ ext \ G_{ext}}{\Gamma, f{:}A \vdash G \ ext \ G_{ext}}$	lthin(f)
thin-R	$\frac{\Gamma \vdash \bot \ ext \ X}{\Gamma \vdash G \ ext \ any(X)}$	rthin
$\wedge$-L	$\frac{\Gamma, a{:}A, b{:}B \vdash G \ ext \ G_{ext}}{\Gamma, c{:}A \wedge B \vdash G \ ext \ spread(c, \lambda a.\lambda b.G_{ext})}$	land(c)
$\wedge$-R	$\frac{\Gamma \vdash A \ ext \ A_{ext} \quad \Gamma \vdash B \ ext \ B_{ext}}{\Gamma \vdash A \wedge B \ ext \ (A_{ext}, B_{ext})}$	rand
$\vee$-L	$\frac{\Gamma, a{:}A \vdash G \ ext \ G^A_{ext} \quad \Gamma, b{:}B \vdash G \ ext \ G^B_{ext}}{\Gamma, c{:}A \vee B \vdash G \ ext \ decide(c, \lambda a.G^A_{ext}, \lambda b.G^B_{ext})}$	lor(c)
$\vee$-R(i)	$\frac{\Gamma \vdash A \ ext \ G^A_{ext}}{\Gamma \vdash A \vee B \ ext \ left(G^A_{ext})}$	ror(left)
$\vee$-R(ii)	$\frac{\Gamma \vdash B ext \ G^B_{ext}}{\Gamma \vdash A \vee B \ ext \ right(G^B_{ext})}$	ror(right)
$\rightarrow$-L	$\frac{\Gamma, b{:}B \vdash G \ ext \ G^B_{ext} \quad \Gamma \vdash A \ ext \ A_{ext}}{\Gamma, f{:}A \rightarrow B \vdash G ext \ G^B_{ext}[f(A_{ext})/b]}$	limp(f)
$\rightarrow$-R	$\frac{\Gamma, a{:}A \vdash B \ ext B_{ext}}{\Gamma \vdash A \rightarrow B \ ext \ \lambda a.B_{ext}}$	rimp
basic	$\Gamma, b : A \vdash A \ ext \ b$	basic
cut	$\frac{\Gamma, a{:}A \vdash B \ ext \ B^A_{ext} \quad \Gamma \vdash A \ ext \ A_{ext}}{\Gamma \vdash B \ ext \ B^A_{ext}[A_{ext}/a]}$	cut(A)

Table E–1: Propositional Calculus Rules

E.2 The Seldon Proof Environment System

The operating systems OYSTER and Seldon have already been presented in Section 10.2. The rules and corresponding witness terms of Seldon are given in Table E–1, with their reference names used in the system in the right hand column; note that they are merely intuitionistic sequent calculus rules with the witness terms added. The witness terms are terms of lambda-calculus, with some extra functions, namely spread, decide,[1] any, substitution, etc.

This system may be extended to cover predicate logic. A universally quantified proposition $\forall x P(x)$ can be thought of in terms of a function that takes any object y in the set quantified over and constructs a witness to show $P(y)$. The witness term of an existentially quantified proposition $\exists x P(x)$ is a pair consisting of firstly, an object from the set quantified over, and secondly, a proof that this object is a member of the appropriate type. Compare with the rules given in Table E–2. One may obtain an extract term for the proof subtree below a node by using the `extract` command at that node.

---

[1]The functions 'spread' and 'decide' are defined by the expressions: $spread((a,b),c) = c(a)(b)$; $decide(left(a),b,c) = b(a)$; $decide(right(a),b,c) = c(a)$.

Name	*Sequent Rule*	*System-Name*
∀-L	$\frac{\Gamma,z{:}A(t)\vdash G \;\; ext\; G_{ext}}{\Gamma,f{:}\forall x.A(x)\vdash G \;\; ext\; G_{ext}[f(y)/z]}$	luni(f,t)
∀-R	$\frac{\Gamma\vdash A(y) \;\; ext\; A^{y}_{ext}}{\Gamma\vdash \forall x.A(x) \;\; ext\; \lambda x.A^{y}_{ext}[x/y]}$	runi(y)
∃-L	$\frac{\Gamma,z{:}A(y)\vdash G \;\; ext\; G_{ext}}{\Gamma,f{:}\exists x.A(x)\vdash G \;\; ext\; spread(f_{ext},\lambda y.\lambda z.G_{ext})}$	lexi(f,y)
∃-R	$\frac{\Gamma\vdash A(t) \;\; ext\; A^{t}_{ext}}{\Gamma\vdash \exists x.A(x) \;\; ext\; (T,A^{t}_{ext})}$	rexi(t)

Table E–2: Predicate Calculus Rules

A classical system is formed by introducing the law of excluded middle: the rule "divided" accepts any consequent of the form $A \vee \neg A$, and allots it the witness term "magic", provided the system is operating in the classical mode.

The Seldon system may be run by loading `/usr/sin/siani/current/code/seldon_update` on top of the OYSTER system. This system has been considerably extended by the author from Matthew's original version. Code for the rules of this extension of Seldon may be found in `/usr/sin/siani/math_reasoning/crab/rules.pl`, which is one of the files comprising `seldon_update`. Rules are written in the form `rule(rulename,conclusion,[hypotheses])`. For example,

```
rule(rand, _ ==> A # B ext (ExA & ExB), [ ==>A ext ExA, ==>B ext ExB]).
```

The law of the excluded middle only works when 'classical' is true, and is represented by:

```
rule(divided, _ ==> A \ ( A => void) ext magic, []) :-
    (    classical
    ;    write('you are not allowed to exclude the middle currently'),
         nl,
         fail
    ).
```

The cut rule is defined as:

```
rule(cut(A),
     H ==> G  ext substitute(ExA,L,ExG),
     [==> A ext ExA,  T ==> G ext ExG]) :-
 syntax0(A),
 sequence_of_labels([L],H),
 set_add(L:A,H,T).
```

Another rule of interest is `rewrite`, which is a tactic which rewrites terms within double square brackets, in such a way that given the sequent `f(k) ==> g(w)`, then `rewrite((f[[x]] ==> g([[y]])` rewrites `k` to `x` and `w` to `y` in the original expression, thus giving `f(x) ==> g(y)`, but only if these rewrites are admissible.

```
rule(rewrite(Exp),
     H ==> OldExp ext G,
         [==> NewExp ext G | RewriteSubgoals]) :-
     get_rewrite_term(Exp, OldExp, NewExp, RewriteSubgoalsD),
           \+ RewriteSubgoalsD = [],
     get_rewrite_subgoals(RewriteSubgoalsD,RewriteSubgoals, H).
```

The code to represent the ω-rule is discussed in the next section. For further details about the system, including how to make definitions, save and manipulate lemmata, unfold terms, etc., see (Matthews, 1989).

E.3 Representation of the ω-rule and its Subgoals

In order to implement the ω-rule, given a goal of the form '$\forall x P(x)$', it is necessary to generate the hypotheses '$P(r)$ for uniform r', and subsequently to enable this subtree to be completed. It is important to determine what a proof using the constructive ω-rule would look like; this is a difficult problem since only one proof at a time may be operated upon in the Seldon system, and infinitely many proofs cannot be dealt with.

This section presents and explains the actual code written in order to implement the ω-rule. The idea is that, given a goal of form `/\ X::A`, one may try the tactic `find_gen_rule(R)` to provide a generalisation. `autogen(Rulelist)`, where `Rulelist` is the general proof of `A`, may be used to apply a cut within Peano Arithmetic (by working out an appropriate cut formula from the general proof), and has an option to complete the tree as far as possible. `Rulelist` may be inputted manually or else provided by the tactic `find_gen_rule(Rulelist)`. `omega_rule(Rulelist)` is a rule in Seldon which closes the prooftree if appropriate using the constructive ω-rule (and thus in a different system to Peano Arithmetic) by means of the interactive system described in Section E.8. If this option is chosen, a cut may still be applied instead by typing `find_cut`, and then cutting in the appropriate formula, as shown in Section E.8.

In summary, the rule "omega_rule" takes us into the interactive system described in Section E.8, in which a general proof may be recursively constructed by the user, whereas "autogen" (which is a tactic rather than a rule) allows the interactive stage to be omitted, thus fully automating the process. If we do not wish to input the general proof by hand using the interactive system, `find_gen_rule` will provide a suitable automated guess.

```
%%% CODE FOR COMPLETION OF PROOF, BY USE OF CONSTRUCTIVE OMEGA RULE

autogen(List):-
autogen(List,finish).
autogen(List,F):-
autogen{List,2,10,F). % alter numbers, as appropriate

% general proof provided manually, or using 'find_gen_rule(R)',
% generalisation required; M and N are numbers
autogen(List,M,N,F):- % set F to ''finish'' if wish to complete the tree
goal(/\X::A),
assert(gen(/\X::A,List)),
pos(K), select(Z), % sets Z = name of current theorem
make_thms2(proof(n,List),M,N), % gives input for third_stage cf 5
third_stage(/\X::A), % checks correctness of genpf cf section 6
find_cut(C), % finds cut by EBG method cf section 7
```

```
select(Z), pos(K), % returns to initial position in Seldon tree
complete_tree(X,C,F). % applies cut rule, and option to complete tree

% find_gen_rule(R) will return (non-checked) general proof R.
% R will be of the form
% gen(P,[rule1(Pos1,T1*),...]).
% ie. it guesses the general proof, R
find_gen_rule(R):-
goal(/\X::A),
generate_indiv_proofs(/\X::A,M,N), % see subsection 4.1
seldon_to_rule(M,N), % see subsection 4.2
generalise_proof(/\X::A,M). % see section 5

% operator definition for constructive omega rule
:- op(850,fy,['//']).

omega_rule(G):-
apply(omega_rule(G)).

% case when general proof provided manually, or using
% 'find_gen_rule(Rulelist)', closed subtree required
% definition of constructive omega rule
% closes subtree if succeeds; otherwise, will not apply
rule(omega_rule(G), H ==> /\X::A ext ExtN, []):-
pos(K), select(Z), % instantiates K and Z
new_system(X,A), !, % switches into system shown in Section 8
clause(generalisation(G),true), % returns correct generalisation
extract_gen(ExtN), % returns ExtN, the extract term
select(Z), pos(K). % return to original position in Seldon
```

E.4 Input or Generation of Individual Proofs

There is a problem of how to input the individual proofs $P(0), P(1), P(2), \ldots$ to the system, for proof environments usually only deal with one proof at a time. There is also a problem how to incorporate these proofs into the main proof. It was desirable to be able to generate these proofs, rather than inputting them, since they might not be available for input.

The following represents the result of running within Clam[2] code (namely `/home/sin1/siani/current/code/codeC`) written to demonstrate the automatic generation and format conversion of proofs.

E.4.1 Generation of Individual Proofs within Seldon

`generate_indiv_proofs(+Formula,+Number1,+Number2)` creates within the Seldon framework and proves using a fairly basic tactic individual proof instances of the given formula `Formula` for the range of numbers from `Number1` to `Number2` inclusive; these are stored as 0proof, 1proof, 2proof etc., and are shown below as

---

[2]See Subsection 3.2.2 for details. Clam is built on top of OYSTER, and contains many extra useful options.

snapshots. The following illustrates its use within the Seldon system.

```
| ?- generate_indiv_proofs(/\x:: (x+x)+x=x+(x+x),0,2).

yes
| ?- select('0proof').

yes
| ?- snapshot.
0proof: [] complete autotactic(idtac)
==> 0+0+0=0+ (0+0)
by rewrite([[rewrite(0)]]+0=0+[[rewrite(0)]])

  [1] complete
  ==> 0+0=0+0
  by rewrite([[rewrite(0)]]=[[rewrite(0)]])

    [1] complete
    ==> 0=0
    by basic

    [2] complete
    ==> 0=0+0
    by basic

    [3] complete
    ==> 0=0+0
    by basic

  [2] complete
  ==> 0=0+0
  by basic

  [3] complete
  ==> 0=0+0
  by basic


yes
| ?- select('1proof').

yes
| ?- snapshot.
1proof: [] complete autotactic(idtac)
==> s(0)+s(0)+s(0)=s(0)+ (s(0)+s(0))
by rewrite(
  ([[rewrite(s(0+s(0)))]]+s(0)=s(0)+[[rewrite(s(0+s(0)))]]))

  [1] complete
  ==> s(0+s(0))+s(0)=s(0)+s(0+s(0))
  by rewrite([[rewrite(s(0+s(0)+s(0)))]]=s(0)+s(0+s(0)))

    [1] complete
    ==> s(0+s(0)+s(0))=s(0)+s(0+s(0))
    by rewrite(s(0+s(0)+s(0))=[[rewrite(s(0+s(0+s(0))))]])

      [1] complete
      ==> s(0+s(0)+s(0))=s(0+s(0+s(0)))
      by rewrite(
        (s([[rewrite(s(0))]]+s(0))=s(0+s([[rewrite(s(0))]]))))

        [1] complete
        ==> s(s(0)+s(0))=s(0+s(s(0)))
        by rewrite(s([[rewrite(s(0+s(0)))]])=s(0+s(s(0))))

          [1] complete
```

```
          ==> s(s(0+s(0)))=s(0+s(s(0)))
          by rewrite(s(s([[rewrite(s(0))]]))=s(0+s(s(0))))

            [1] complete
            ==> s(s(s(0)))=s(0+s(s(0)))
            by rewrite(s(s(s(0)))=s([[rewrite(s(s(0)))]]))

              [1] complete
              ==> s(s(s(0)))=s(s(s(0)))
              by basic

              [2] complete
              ==> s(s(0))=0+s(s(0))
              by basic

            [2] complete
            ==> s(0)=0+s(0)
            by basic

          [2] complete
          ==> s(0+s(0))=s(0)+s(0)
          by basic

        [2] complete
        ==> s(0)=0+s(0)
        by basic

        [3] complete
        ==> s(0)=0+s(0)
        by basic

      [2] complete
      ==> s(0+s(0+s(0)))=s(0)+s(0+s(0))
      by basic

    [2] complete
    ==> s(0+s(0)+s(0))=s(0+s(0))+s(0)
    by basic

  [2] complete
  ==> s(0+s(0))=s(0)+s(0)
  by basic

  [3] complete
  ==> s(0+s(0))=s(0)+s(0)
  by basic


yes
| ?- select('2proof').

yes
| ?- snapshot.
2proof: [] complete autotactic(idtac)
==> s(s(0))+s(s(0))+s(s(0))=s(s(0))+ (s(s(0))+s(s(0)))
by rewrite(
  ((([[rewrite(s(s(0)+s(s(0))))]]+s(s(0)))
    = (s(s(0))+[[rewrite(s(s(0)+s(s(0))))]])))

  [1] complete
  ==> s(s(0)+s(s(0)))+s(s(0))=s(s(0))+s(s(0)+s(s(0)))
  by rewrite(
    ([[rewrite(s(s(0)+s(s(0))+s(s(0))))]]
      = (s(s(0))+s(s(0)+s(s(0))))))

    [1] complete
    ==> s(s(0)+s(s(0))+s(s(0)))=s(s(0))+s(s(0)+s(s(0)))
```

```
by rewrite(
  (s([[rewrite(s(0+s(s(0))))]]+s(s(0)))
    = (s(s(0))+s([[rewrite(s(0+s(s(0))))]])))))

  [1] complete
  ==> s(s(0+s(s(0)))+s(s(0)))=s(s(0))+s(s(0+s(s(0))))
  by rewrite(
    (s([[rewrite(s(0+s(s(0))+s(s(0))))]])
      = (s(s(0))+s(s(0+s(s(0))))))

    [1] complete
    ==> s(s(0+s(s(0))+s(s(0))))=s(s(0))+s(s(0+s(s(0))))
    by rewrite(
      (s(s(0+s(s(0))+s(s(0))))
        = [[rewrite(s(s(0)+s(s(0+s(s(0))))))]]))

      [1] complete
      ==> s(s(0+s(s(0))+s(s(0))))=s(s(0)+s(s(0+s(s(0)))))
      by rewrite(
        (s(s(0+s(s(0))+s(s(0))))
          = s([[rewrite(s(0+s(s(0+s(___)))))]])))

        [1] complete
        ==> (s(s(0+s(s(0))+s(s(0))))
            = s(s(0+s(s(0+s(s(0))))))
        by rewrite(
          (s(s([[rewrite(s(s(0)))]]+s(s(0))))
            = s(s(0+s(s([[rewrite(s(s(0)))]])))))

          [1] complete
          ==> s(s(s(s(0))+s(s(0))))=s(s(0+s(s(s(s(0))))))
          by rewrite(
            (s(s([[rewrite(s(s(0)+s(s(0))))]]))
              = s(s(0+s(s(s(s(0)))))))

            [1] complete
            ==> s(s(s(s(0)+s(s(0)))))=s(s(0+s(s(s(s(0))))))
            by rewrite(
              (s(s(s(s([[rewrite(s(0+s(s(0))))]])))
                = s(s(0+s(s(s(s(0)))))))

              [1] complete
              ==> s(s(s(s(0+s(s(0))))))=s(s(0+s(s(s(s(0))))))
              by rewrite(
                (s(s(s(s(s(0))))))
                  = s(s([[rewrite(s(s(s(s(0)))))]])))

                [1] complete
                ==> s(s(s(s(s(0))))))=s(s(s(s(s(0))))))
                by basic

                [2] complete
                ==> s(s(s(0))))=0+s(s(s(0))))
                by basic

              [2] complete
              ==> s(s(0))=0+s(s(0))
              by basic

            [2] complete
            ==> s(0+s(s(0)))=s(0)+s(s(0))
```

```
                by basic

              [2] complete
              ==> s(s(0)+s(s(0)))=s(s(0))+s(s(0))
              by basic

            [2] complete
            ==> s(s(0))=0+s(s(0))
            by basic

            [3] complete
            ==> s(s(0))=0+s(s(0))
            by basic

          [2] complete
          ==> s(0+s(s(0+s(s(0)))))=s(0)+s(s(0+s(s(0))))
          by basic

        [2] complete
        ==> s(s(0)+s(s(0+s(s(0)))))=s(s(0))+s(s(0+s(s(0))))
        by basic

      [2] complete
      ==> s(0+s(s(0))+s(s(0)))=s(0+s(s(0)))+s(s(0))
      by basic

    [2] complete
    ==> s(0+s(s(0)))=s(0)+s(s(0))
    by basic

    [3] complete
    ==> s(0+s(s(0)))=s(0)+s(s(0))
    by basic

  [2] complete
  ==> s(s(0)+s(s(0))+s(s(0)))=s(s(0)+s(s(0)))+s(s(0))
  by basic

[2] complete
==> s(s(0)+s(s(0)))=s(s(0))+s(s(0))
by basic

[3] complete
==> s(s(0)+s(s(0)))=s(s(0))+s(s(0))
by basic

yes
```

E.4.2 Conversion of Proofs from Seldon to CORE Representation

Note that the 'rewrite' notation above is used to represent within Seldon the process of rewriting using a rewrite rule; within the CORE representation, a more explicit notation is used denoting particular rules used, namely "apply rule R at subposition P, number of times k". In the case below, `rule1` is $s(x)+y \rightarrow s(x+y)$, and `rule2` is $0+y \rightarrow y$.

The representation for each individual proof for $P(J)$ would be of the form:

$$proof(J, [Rule1_J(Pos1_J, T1_J), Rule2_J(Pos1_J, T2_J), \ldots])$$

This is a predicate, its first argument being the natural number J, and the other argument being an ordered list of predicates such that $RuleK_J$ is the Kth rule to be applied in the proof, and is applied TK_J times, with $PosK_J$ specifying the position at which it is applied. Note that we may have $RuleA_J = RuleB_J$.

`seldon_to_rule(+Number1,+Number2)` converts the proofs stored as 0proof, 1proof and 2proof to a rule formulation used to enable manipulation of a general proof, namely 'proof(J,Rulelist)' for `Number1` $\leq J \leq$ `Number2`. The results are associated with an appropriate integer within the range from `Number1` to `Number2` inclusive, and asserted into the database, as shown below.

```
| ?- seldon_to_rule(0,3).

yes
| ?- proof(X,Y).

X = 0,
Y = [rule2([1,1],1),rule2([2,2],1),rule2([1],1),rule2([2],1)] ;
X = 1,
Y = [rule1([1,1],1),rule1([2,2],1),rule1([1],1),rule1([2],1),
rule2([1,1,1],1),rule2([1,2,1,2],1),rule1([1,1],1),rule2([1,1,1],1),
rule2([1,2],1)] ;
X = 2,
Y = [rule1([1,1],1),rule1([2,2],1),rule1([1],1),rule1([1,1,1],1),
rule1([1,2,2],1),rule1([1,1],1),rule1([2],1),rule1([1,2],1),
rule2([1,1,1,1],1),rule2([1,1,2,1,1,2],1),rule1([1,1,1],1),
rule1([1,1,1,1],1),rule2([1,1,1,1,1],1),rule2([1,1,2],1)] ;
X = 3,
Y = [rule1([1,1],1),rule1([2,2],1),rule1([1],1),rule1([1,1,1],1),
rule1([1,2,2],1),rule1([1,1],1),rule1([1,1,1,1],1),rule1([1,1,2,2],1),
rule1([1,1,1],1),rule1([2],1),rule1([1,2],1),rule1([1,1,2],1),
rule2([1,1,1,1,1],1),rule2([1,1,1,2,1,1,1,2],1),rule1([1,1,1,1],1),
rule1([1,1,1,1,1],1),rule1([1,1,1,1,1,1],1),rule2([1,1,1,1,1,1,1],1),
rule2([1,1,1,2],1)] ;

no
```

E.5 Provision of a General Proof

In Section 10.3 I considered how the general proofs described in Chapter 6 might be represented from an implementational point of view. The algorithm presented in that section *automatically* recognises a general pattern, and hence forms the basis for an implementation, upon input of individual proofs for $P(1), P(2), \ldots$. `find_gen_rule(GENPF)` returns the non-checked general proof `GENPF`, and follows this algorithm. The code is to be found in `/home/sin1/siani/current/code/newcode`. This code uniformly generates individual proofs in CORE (`make_thms2/3`), guesses an initial generalisation from an individual 'starting' proof (`generalise_proof/2`), and also updates the general proof using the other individual proofs, until the general proof seems to have reached a stable form (`inspect_rep/3`). The general representation of the proofs is asserted into the database as 'gen(Sequent,Rulelist)'. If the above code is run within Clam, after having loaded the Seldon system, a general proof may be provided as follows:

EXAMPLE ONE

```
| ?- make_thms2(proof(n,[rule1([],n),rule2([2],3),rule1([1],2*n)],2,7).
% note the general proof
yes % we wish to produce
| ?- proof(X,Y).

X = 2, % cf. code A -
Y = [rule1([],2),rule2([2],3),rule1([1],4)] ? % method of generating
X = 3, % individual proofs
Y = [rule1([],3),rule2([2],3),rule1([1],6)] ?
X = 4,
Y = [rule1([],4),rule2([2],3),rule1([1],8)] ?
X = 5,
Y = [rule1([],5),rule2([2],3),rule1([1],10)] ?
X = 6,
Y = [rule1([],6),rule2([2],3),rule1([1],12)] ?
X = 7,
Y = [rule1([],7),rule2([2],3),rule1([1],14)] ?

no
| ?- generalise(2). % produces first general
% proof (see below)
yes % note that this general
| ?- gen(X). % proof is not the final one

X = [rule1([],n),rule2([2],n+1),rule1([1],2*n)] ?

no
| ?- inspect_rep(3,0). % recursive part

yes
| ?- gen(X). % final general proof
% produced - note this is
X = [rule1([],n),rule2([2],3),rule1([1],2*n)] ? % the correct one!

no
| ?- halt.
```

EXAMPLE TWO

```
| ?- make_thms2(proof(n,[rule1([],3*n),rule2([2],n),rule3([],1),rule2([1,1],
n+2)]),5,10). % note the general proof given
% from which the individual proofs
yes % will be generated
| ?- proof(X,Y).

X = 5, % the individual proofs generated
Y = [rule1([],15),rule2([2],5),rule3([],1),rule2([1,1],7)] ? ;
X = 6,
Y = [rule1([],18),rule2([2],6),rule3([],1),rule2([1,1],8)] ? ;
X = 7,
Y = [rule1([],21),rule2([2],7),rule3([],1),rule2([1,1],9)] ? ;
X = 8,
Y = [rule1([],24),rule2([2],8),rule3([],1),rule2([1,1],10)] ? ;
X = 9,
Y = [rule1([],27),rule2([2],9),rule3([],1),rule2([1,1],11)] ? ;
X = 10,
Y = [rule1([],30),rule2([2],10),rule3([],1),rule2([1,1],12)] ? ;

no
| ?- generalise(5). % cf. Code B

yes
```

```
| ?- gen(X).

X = [rule1([],3*n),rule2([2],n),rule3([],n+ -4),rule2([1,1],n+2)] ? ;

no
| ?- inspect_rep(6,0). % cf. Code C

yes
| ?- gen(X). % this is the general proof
% returned - the correct one
X = [rule1([],3*n),rule2([2],n),rule3([],1),rule2([1,1],n+2)] ? ;

no
```

E.6 Checking Correctness of the General Proof

Section 10.3.1 has already considered how it could be known whether the general proof is the right one — in other words, how one could know that the next individual proof example selected would not falsify it? This involves meta-level reasoning as a check to ensure that the general method is the required one. `third_stage(GOAL)` checks the correctness of the general proof of `GOAL`, and follows the algorithm given in Section 10.3.1. It outputs the lines of the 'general proof' and also the guessed formulae used.

E.6.1 Examples

The relevant additional code for this stage is found in `siani/current/code/stage3` (which is encorporated into `siani/current/code/new_system2` — see Section E.8) and which carries out the process described above.

```
| ?- make_thms2(proof(n,[rule1([1,1],n),rule1([2],n),rule2([2,2,1,1],1),rule2
([2,2,2],1),rule1([1],n)]),2,7).
% note that this is the general proof we wish to produce
yes
| ?- proof(X,Y). % these are the (inputted) proofs

X = 2,
Y =
[rule1([1,1],2),rule1([2],2),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],2)] ?
X = 3,
Y =
[rule1([1,1],3),rule1([2],3),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],3)] ?
X = 4,
Y =
[rule1([1,1],4),rule1([2],4),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],4)] ?
X = 5,
Y =
[rule1([1,1],5),rule1([2],5),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],5)] ?
X = 6,
Y =
[rule1([1,1],6),rule1([2],6),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],6)] ?
X = 7,
Y =
[rule1([1,1],7),rule1([2],7),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],7)] ?
```

```
no
| ?- generalise(2). % produces first general proof - note that this
% general proof is not the final one
yes
| ?- gen(X). % general proof produced

X = [rule1([1,1],n),rule1([2],n),rule2([2,2,1,1],n+
-1),rule2([2,2,2],n+ -1),rule1([1],n)] ?

no
| ?- inspect_rep(3,0). % recursive part, to produce general proof

yes
| ?- gen(X). % final general proof produced

X =
[rule1([1,1],n),rule1([2],n),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],n)] ?

no % note this is in fact the correct one!
| ?- third_stage(/\x:: (x+x)+x=x+(x+x)). % checking this is the correct
Subrule used: s^(m,s^(n-m,0)+s(s^(n-1,0))) % general proof
New equation is, using rule1: s^(n,0+s(s^(n-1,0)))+s^(n,0)=s^(n,0)+(s^(n,0)+s^(n,0))
Subrule used: s^(m,s^(n-m,0)+(s(s^(n-1,0))+s(s^(n-1,0))))
New equation is, using rule1:
s^(n,0+s(s^(n-1,0)))+s^(n,0)=s^(n,0+(s(s^(n-1,0))+s(s^(n-1,0))))
New equation is, using rule2:
s^(n,s(s^(n-1,0)))+s^(n,0)=s^(n,0+(s(s^(n-1,0))+s(s^(n-1,0))))
New equation is, using rule2:
s^(n,s(s^(n-1,0)))+s^(n,0)=s^(n,s(s^(n-1,0))+s(s^(n-1,0)))
Subrule used: s^(m,s^(n-m,s(s^(n-1,0)))+s(s^(n-1,0)))
New equation is, using rule1:
s^(n,s(s^(n-1,0))+s(s^(n-1,0)))=s^(n,s(s^(n-1,0))+s(s^(n-1,0)))
Final equation is: s^(n,s(s^(n-1,0))+s(s^(n-1,0)))=s^(n,s(s^(n-1,0))+s(s^(n-1,0)))

Correct general proof is:
[rule1([1,1],n),rule1([2],n),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],n)]
yes   % the general proof was shown to be a correct one
```

A Further Example

```
| ?- make_thms2(proof(n,[rule1([1],n),rule1([2],3),rule2([2,2,1],1),rule2([1,
1,1,2],1)]),2,7).

yes
| ?- proof(X,Y).

X = 2,
Y = [rule1([1],2),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)] ?
X = 3,
Y = [rule1([1],3),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)] ?
X = 4,
Y = [rule1([1],4),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)] ?
X = 5,
Y = [rule1([1],5),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)] ?
X = 6,
Y = [rule1([1],6),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)] ?
X = 7,
Y = [rule1([1],7),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)] ?

no
| ?- generalise(2).

yes
| ?- gen(X).

X = [rule1([1],n),rule1([2],n+1),rule2([2,2,1],n+ -1),rule2([1,1,1,2],n+ -1)] ?
```

```
yes
| ?- inspect_rep(3,0).

yes
| ?- gen(X).

X = [rule1([1],n),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)] ?

yes

| ?- third_stage(/\x:: x+3=3+x).
Subrule used: s^(m,s^(n-m,0)+s(s(s(0))))
New equation is, using rule1: s^(n,0+s(s(s(0))))=s(s(s(0)))+s^(n,0)
New equation is, using rule1 applied 3 times: s^(n,0+s(s(s(0))))=s(s(s(0+s^(n,0))))
New equation is, using rule2 applied 1 times: s^(n,s(s(s(0))))=s(s(s(0+s^(n,0))))
New equation is, using rule2 applied 1 times: s^(n,s(s(s(0))))=s(s(s(s^(n,0))))
Final equation is: s^(n,s(s(s(0))))=s(s(s(s^(n,0))))

Correct general proof is:
[rule1([1],n),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)]
yes
```

E.7 Generalisation

The generalisation process comprising the suggestion of cut formulae for arithmetic has been implemented. This is done by explanation-based generalisation, as described in Chapters 7 and 8, following the stages described above, namely the provision of a correct general proof. The transcript in the following section illustrates the user interface. The code for finding the cut formula is located in `siani/current/code/find_cut4`.

E.8 Interactive System

The system allows interaction with the user to define the general proof recursively, or else to produce the general proof automatically. This involves application of rewrite rules, and a tree representation. Code for the interactive system is located in `/home/sin1/siani/current/code/new_system2`. This system encorporates all the code mentioned above, and follows the flow pattern in the following subsection.

Allowance is made for branching proofs by using a tree structure for the proof, as may be seen by the position information in the general proof specifying such tree nodes. A simple rewrite rule will be of the form $P \Rightarrow Q$ (for example $s(x) + y \Rightarrow s(x + y)$). A branching rule will be of the form $P \Rightarrow Q_1, \ldots Q_n$ and may be represented internally by a tree from Seq at position $[k]$ to subnodes A_i at tree positions $[k, i]$, for $1 \leq i \leq n$. In the case of the rule being applied to subgoals recursively $f(m)$ times, where m is a meta-variable, this is represented internally by a tree from position $[k]$ to subnodes at tree positions $[k, f(m), i]$ for $1 \leq i \leq n$. By these means it is possible to express any finite tree of applications of a rewrite rule.

A general description of the interactive system has already been given in Chapter 10. The following subsections illustrate the flow of the system, and provide transcripts illustrating how it may be used.

E.8.1 CORE Processes

Type "omega-rule"
$\rightarrow$ goto **A**

A: Input stored rules + Prompt for meta-variable eg. $n \rightarrow$ goto **B**

B: General formula displayed (cf. $s^n(0)$) + set current node to [] $\rightarrow$ goto **C**

C: Prompt for full general proof to be given $\rightarrow$ YES: goto **G**; NO: goto **D**

G: Input full generalisation. Check general proof by working out new subgoals and justification. $\rightarrow$ correct: goto **K**; incorrect: goto **N**

N: Output "wrong general proof" message. Goto **C** or halt.

D: Ask for new current node required, pos(X). Prompt for rule application, or finished tree belief. $\rightarrow$ branching rule: goto **P**; constant number of applications of rule: goto **F**; rule applied function of n times: goto **E**; finished tree: goto **M**.

P: Branching rule case. Representation is "branch(rule(Subpos of expression, No of times applied), List of nodes eg. [1,3] describing subterms to which to apply more than once (if appropriate), [[Subtree (1st node)],[Subtree (2nd node)], ..., [Subtree (nth node)]])". Subdivision according to whether the rule is applied a constant or a parametrised number of times, as with **D**.

E: The rule must be applied a parametrised number of times. Prompt for final subgoal. $\rightarrow$ Given: goto **Z**; not given: goto **Y**

Y: work out new subgoal $\rightarrow$ goto **Z**

Z: return justification (by induction on n) $\rightarrow$ works: goto **X**; does not work: goto **H**

X: display new subgoal. Assert position and information about node and subnodes. Display new subgoal. Goto **D**.

H: Error message plus justification. Goto **D**.

F: The rule must be applied a constant number of times. Work out new subgoal. → works: goto **X**; does not work: goto **H**

M: Check to see if the proof is finished: equality check on all subgoals. → not equal: goto **H**; equal: goto **K**

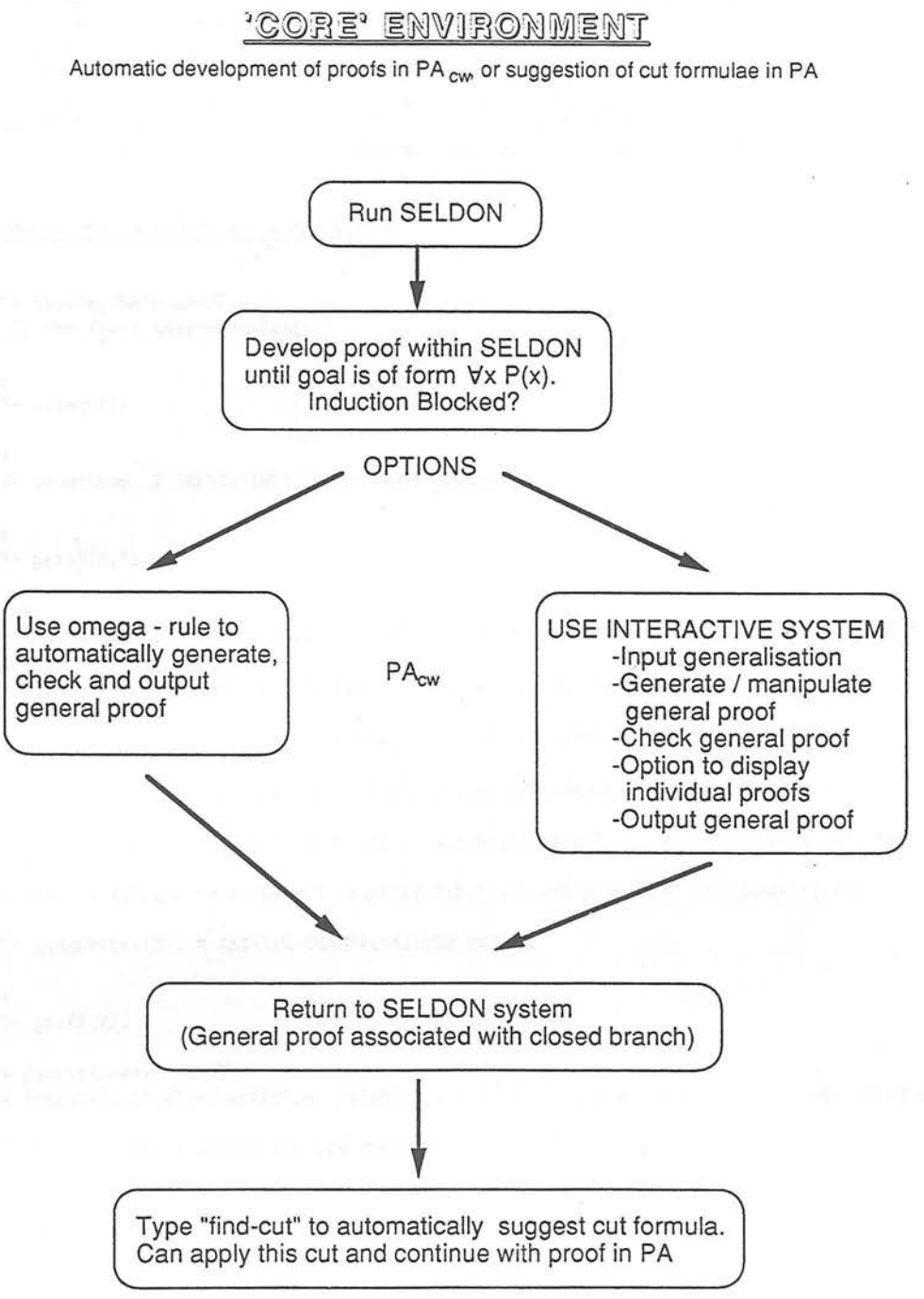
K: output (pretty printed) general proof (by means of asserted information) → goto **L**

L: Prompt for display of individual proof. → yes: goto **Q**; no: goto **O**

Q: Work out and display. Goto **L**.

O: Return to Seldon system. Close subtree, associate general proof with the node and provide an appropriate extract term.

E.8.2 Structure of CORE



E.8.3 Transcript of CORE environment, illustrating generation of general proofs and automatic provision of cut formula via the explanation-based generalisation method

```
% loading file /home/ossian1/siani/current/code/demo
%  foreign file /usr/local/src/quintus3/generic/qplib3.1.2/library/sun4-4/libpl.so loaded
% demo loaded in module user, 1.883 sec 584,040 bytes
Quintus Prolog Release 3.1.2 (Sun-4, SunOS 4.1)
Copyright (C) 1990, Quintus Corporation.  All rights reserved.
2100 Geng Road, Palo Alto, California U.S.A. (415) 813-3800
CLaM Proof Planner Version 1.5 (libraries only) (7/3/93 20:24)
CLaM Proof Planner Version 1.5: (BETA-testing) (7/3/93 20:54)
```

Example One: $\forall x\ (x+x)+x = x+(x+x)$

```
| ?- create_thm(s,user).
|: [] ==> /\x:: (x+x)+x=x+(x+x).

yes
| ?- select(s).

yes
| ?- inputthms. % INPUTS INDIVIDUAL PROOFS BELOW

yes
| ?- proof(X,Y).

X = 2,
Y = [rule1([1,1],2),rule1([2],2),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],2)] ;
X = 3,
Y = [rule1([1,1],3),rule1([2],3),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],3)] ;
X = 4,
Y = [rule1([1,1],4),rule1([2],4),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],4)] ;
X = 5,
Y = [rule1([1,1],5),rule1([2],5),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],5)] ;
X = 6,
Y = [rule1([1,1],6),rule1([2],6),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],6)] ;
X = 7,
Y = [rule1([1,1],7),rule1([2],7),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],7)]

| ?- generalise(2). % INITIAL GENERALISATION GUESSED

yes
| ?- gen(X,Y).

X = /\x::x+x+x=x+ (x+x),
Y = [rule1([1,1],n),rule1([2],n),rule2([2,2,1,1],n+ -1),rule2([2,2,2],n+ -1),rule1([1],n)]

| ?- inspect_rep. % GENERALISATION UPDATED

yes
| ?- gen(X,Y).

X = /\x::x+x+x=x+ (x+x),
Y = [rule1([1,1],n),rule1([2],n),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],n)]

| ?- display. % NODE IN SELDON SYSTEM
s: [] incomplete autotactic(idtac)
==> /\x::x+x+x=x+ (x+x)
by _

yes
```

```
| ?- induce. % USE INDUCTION RULE

yes
| ?- snapshot.
s: [] partial autotactic(idtac)
==> /\x::x+x+x=x+ (x+x)
by induction(x)

  [1] incomplete
  ==> 0+0+0=0+ (0+0)
  by _

  [2] incomplete
  v0. x+x+x=x+ (x+x)
  ==> s(x)+s(x)+s(x)=s(x)+ (s(x)+s(x))
  by _ % THIS BRANCH IS EVENTUALLY BLOCKED

yes
| ?- omega_rule(_). % CALLS INTERACTIVE SYSTEM
Which meta-variable do you wish to use?
If not known, type n.
|: n.
General assertion (to be proved) is s^ (n,0)+s^ (n,0)+s^ (n,0)=s^ (n,0)+ (s^ (n,0)+s^ (n,0)).

Do you wish to input a full generalisation? (yes/no)
|: yes. % INPUTTING FULL GENERALISATION

Input general proof in the form of a list with members:
Rule(Pos of Subexpression, No of times applied)
or branch(Rule,Pos,No,[[S1],...,[SK]])
|: inputgen. % INPUTS THE GENERALISATION PRODUCED ABOVE

Original expression is, at position []:
s^ (n,0)+s^ (n,0)+s^ (n,0)=s^ (n,0)+ (s^ (n,0)+s^ (n,0))

Subrule used: s^ (m,s^ (n-m,0)+s^ (n,0))
New equation is, at position [n,1] using rule1 applied n times:
s^ (n,0+s^ (n,0))+s^ (n,0)=s^ (n,0)+ (s^ (n,0)+s^ (n,0))
Subrule used: s^ (m,s^ (n-m,0)+ (s^ (n,0)+s^ (n,0)))
New equation is, at position [n,1,n,1] using rule1 applied n times:
s^ (n,0+s^ (n,0))+s^ (n,0)=s^ (n,0+ (s^ (n,0)+s^ (n,0)))
New equation is, at position [n,1,n,1,1] using rule2 applied 1 times:
s^ (n,s^ (n,0))+s^ (n,0)=s^ (n,0+ (s^ (n,0)+s^ (n,0)))
New equation is, at position [n,1,n,1,1,1] using rule2 applied 1 times:
s^ (n,s^ (n,0))+s^ (n,0)=s^ (n,s^ (n,0)+s^ (n,0))
Subrule used: s^ (m,s^ (n-m,s^ (n,0))+s^ (n,0))
New equation is, at position [n,1,n,1,1,1,n,1] using rule1 applied n times:
s^ (n,s^ (n,0)+s^ (n,0))=s^ (n,s^ (n,0)+s^ (n,0))
Final equation is: s^ (n,s^ (n,0)+s^ (n,0))=s^ (n,s^ (n,0)+s^ (n,0))
Correct general proof.

Type "yes" to see an individual proof, type "no" to
exit the system.
|: yes.

Input index of proof:
|: 3.

Individual proof rules to be applied are:
[rule1([1,1],3),rule1([2],3),rule2([2,2,1,1],1),rule2([2,2,2],1),rule1([1],3)]

Would you like the individual proof, if possible?
|: yes.

Initial sequent is, at tree-position []:
s^ (3,0)+s^ (3,0)+s^ (3,0)=s^ (3,0)+ (s^ (3,0)+s^ (3,0))
At position [1,1,1,1] the sequent is
```

```
s^ (3,0+s^ (3,0))+s^ (3,0)=s^ (3,0)+ (s^ (3,0)+s^ (3,0)) using rule1([1,1],3)
At position [1,1,1,1,1,1,1,1] the sequent is
s^ (3,0+s^ (3,0))+s^ (3,0)=s^ (3,0+ (s^ (3,0)+s^ (3,0))) using rule1([2],3)
At position [1,1,1,1,1,1,1,1,1] the sequent is
s^ (3,s^ (3,0))+s^ (3,0)=s^ (3,0+ (s^ (3,0)+s^ (3,0))) using rule2([2,2,1,1],1)
At position [1,1,1,1,1,1,1,1,1] the sequent is
s^ (3,s^ (3,0))+s^ (3,0)=s^ (3,s^ (3,0)+s^ (3,0)) using rule2([2,2,2],1)
At position [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1] the sequent is
s^ (3,s^ (3,0)+s^ (3,0))=s^ (3,s^ (3,0)+s^ (3,0)) using rule1([1],3)

Type "yes" to see an individual proof, type "no" to
exit the system.
|: no.

yes
| ?- display. % BRANCH COMPLETED BY OMEGA RULE
s: [] complete autotactic(idtac)
==> /\x::x+x+x=x+ (x+x)
by omega_rule(
  [rule1([1,1],n),rule1([2],n),rule2([2,2,1,1],1),rule2([2,2,2],1),
   rule1([1],n)])

yes
| ?- retract. % REMOVE ASSERTED INFORMATION

yes
| ?- omega_rule(_). % USER WISHES TO INPUT THE GENERAL PROOF
Which meta-variable do you wish to use?
If not known, type n.
|: n.
General assertion (to be proved) is
s^ (n,0)+s^ (n,0)+s^ (n,0)=s^ (n,0)+ (s^ (n,0)+s^ (n,0)).

Do you wish to input a full generalisation? (yes/no)
|: no.

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([1,1],n).

Current node is: []
Do you wish to select another position? (yes/no)
|: no.

Subexpression to be altered is: s^ (n,0)+s^ (n,0)
Do you wish to input subexpression after rule application? (yes/no)
|: no.

Subrule used: s^ (m,s^ (n-m,0)+s^ (n,0))
New equation is, using rule1([1,1],n):
s^ (n,0+s^ (n,0))+s^ (n,0)=s^ (n,0)+ (s^ (n,0)+s^ (n,0))

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([2],n).

Current node is: [n,1]
Do you wish to select another position? (yes/no)
|: no.

Subexpression to be altered is: s^ (n,0)+ (s^ (n,0)+s^ (n,0))
Do you wish to input subexpression after rule application? (yes/no)
|: yes.

Input subexpression: s^(n,0)+s^(n,0).

Trying to induce:
```

```
The base case of (meta)induction is to prove
y+s^ (0,0)=y+ (s^ (0,0)+s^ (0,0))

Base case does not follow by basic "successor" definitions.
The step case of (meta)induction is to prove
y+s^ (s(n),0)=y+ (s^ (s(n),0)+s^ (s(n),0)) assuming y+s^ (n,0)=y+ (s^ (n,0)+s^ (n,0))

The meta-induction does not work.
The induction did not succeed.
The subexpression given was not the correct one.

Subexpression to be altered is: s^ (n,0)+ (s^ (n,0)+s^ (n,0))
Do you wish to input subexpression after rule application? (yes/no)
|: yes.

Input subexpression: s^(n,0+ (s^(n,0)+s^(n,0))).

Trying to induce:
The base case of (meta)induction is to prove
s^ (0,0+y)=s^ (0,0)+y

Base clase follows by definitions of successor function.

The step case of (meta)induction is to prove
s^ (s(n),0+y)=s^ (s(n),0)+y assuming s^ (n,0+y)=s^ (n,0)+y

Using rule1 on RHS, we obtain: s^ (s(n),0+y)=s(s^ (n,0)+y)
Fertilising with the hypothesis, we obtain:
s^ (s(n),0+y)=s(s^ (n,0+y))
This is an equality using the "successor" definitions.

The meta-induction worked, showing that the subexpression was
a correct one.

Hence the new equation is, using rule1([2],n):
s^ (n,0+s^ (n,0))+s^ (n,0)=s^ (n,0+ (s^ (n,0)+s^ (n,0)))
Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule2([2,2,1,1],1).

Current node is: [n,1,n,1]
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule2 applied 1 time(s):
s^ (n,s^ (n,0))+s^ (n,0)=s^ (n,0+ (s^ (n,0)+s^ (n,0))) at tree position [n,1,n,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule2([2,2,2],1).

Current node is: [n,1,n,1,1]
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule2 applied 1 time(s):
s^ (n,s^ (n,0))+s^ (n,0)=s^ (n,s^ (n,0)+s^ (n,0)) at tree position [n,1,n,1,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([1],n).

Current node is: [n,1,n,1,1,1]
Do you wish to select another position? (yes/no)
|: no.

Subexpression to be altered is: s^ (n,s^ (n,0))+s^ (n,0)
```

```
Do you wish to input subexpression after rule application? (yes/no)
|: no.

Subrule used: s^ (m,s^ (n-m,s^ (n,0))+s^ (n,0))
New equation is, using rule1([1],n):
s^ (n,s^ (n,0)+s^ (n,0))=s^ (n,s^ (n,0)+s^ (n,0))

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: finished_tree.

General proof:

At position [] the sequent is s^ (n,0)+s^ (n,0)+s^ (n,0)=
s^ (n,0)+ (s^ (n,0)+s^ (n,0))
At position [n,1] the sequent is s^ (n,0+s^ (n,0))+s^ (n,0)=
s^ (n,0)+ (s^ (n,0)+s^ (n,0)) using rule rule1([1,1],n)
At position [n,1,n,1] the sequent is s^ (n,0+s^ (n,0))+s^ (n,0)=
s^ (n,0+ (s^ (n,0)+s^ (n,0))) using rule rule1([2],n)
At position [n,1,n,1,1] the sequent is s^ (n,s^ (n,0))+s^ (n,0)=
s^ (n,0+ (s^ (n,0)+s^ (n,0))) using rule rule2([2,2,1,1],1)
At position [n,1,n,1,1,1] the sequent is s^ (n,s^ (n,0))+s^ (n,0)=
s^ (n,s^ (n,0)+s^ (n,0)) using rule rule2([2,2,2],1)
At position [n,1,n,1,1,1,n,1] the sequent is s^ (n,s^ (n,0)+s^ (n,0))=
s^ (n,s^ (n,0)+s^ (n,0)) using rule rule1([1],n)

Type "yes" to see an individual proof, type "no" to
exit the system.
|: no.

yes
| ?- display. % SELDON BRANCH COMPLETED USING OMEGA RULE
s: [] complete autotactic(idtac)
==> /\x::x+x+x=x+ (x+x)
by omega_rule(
  [rule1([1,1],n),rule1([2],n),rule2([2,2,1,1],1),rule2([2,2,2],1),
   rule1([1],n)])

yes
| ?- find_cut. % AUTOMATICALLY PRODUCES CUT FORMULA
% - EBG WORKING PRINTED OUT
The expression, after applying rule1 n time(s), is:
fn0(s^ (n,_7223+_7217),_7272,_7273)=_7139
The expression, after applying rule1 n time(s), is:
fn0(s^ (n,_7223+_7217),_7272,_7273)=s^ (n,_7450+_7444)
The expression, after applying rule2 1 time(s), is:
fn0(s^ (n,_7217),_7272,_7273)=s^ (n,_7450+_7444)
The expression, after applying rule2 1 time(s), is:
fn0(s^ (n,_7217),_7272,_7273)=s^ (n,_7444)
The expression, after applying rule1 n time(s), is:
s^ (n,_7217+_7272)=s^ (n,_7444)
The generalisation is /\x::x+v0+v1=x+ (v0+v1)

yes
| ?- cut(/\v0:: (/\v1:: (/\x:: x+v0+v1=x+(v0+v1)))).
% THE SUGGESTED CUT MAY BE APPLIED
yes
| ?- d.
==> /\x::x+x+x=x+ (x+x)
by cut(/\v0:: /\v1:: /\x::x+v0+v1=x+ (v0+v1))
incomplete ==> /\v0:: /\v1:: /\x::x+v0+v1=x+ (v0+v1)
incomplete ==> /\x::x+x+x=x+ (x+x)

yes
| ?- display.
s: [] partial autotactic(idtac)
==> /\x::x+x+x=x+ (x+x)
```

```
by cut(/\v0:: /\v1:: /\x::x+v0+v1=x+ (v0+v1))

  [1] incomplete
  ==> /\v0:: /\v1:: /\x::x+v0+v1=x+ (v0+v1)

  [2] incomplete
  v0. /\v0:: /\v1:: /\x::x+v0+v1=x+ (v0+v1)
  ==> /\x::x+x+x=x+ (x+x)

yes
% THIS PROOF MAY EASILY BE COMPLETED
```

Example Two: $\forall x\ x + 3 = 3 + x$

```
| ?- create_thm(t,user).
|: [] ==> /\x:: x+s(s(s(0)))=s(s(s(0)))+x.

yes
| ?- select(t).

yes
| ?- d.
==> /\x::x+s(s(s(0)))=s(s(s(0)))+x
by _

yes
| ?- omega_rule(_).
Which meta-variable do you wish to use?
If not known, type n.
|: n.
General assertion (to be proved) is s^ (n,0)+s(s(s(0)))=s(s(s(0)))+s^ (n,0).

Do you wish to input a full generalisation? (yes/no)
|: no.

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([1],n).

Current node is: []
Do you wish to select another position? (yes/no)
|: no.

Subexpression to be altered is: s^ (n,0)+s(s(s(0)))
Do you wish to input subexpression after rule application? (yes/no)
|: no.

Subrule used: s^ (m,s^ (n-m,0)+s(s(s(0))))
New equation is, using rule1([1],n):s^ (n,0+s(s(s(0))))=s(s(s(0)))+s^ (n,0)

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([2],3).

Current node is: [n,1]
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule1 applied 3 time(s): s^ (n,0+s(s(s(0))))=
s(s(s(0+s^ (n,0)))) at tree position [n,1,1,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule2([2,2,1],1).

Current node is: [n,1,1,1,1]
```

```
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule2 applied 1 time(s): s^ (n,s(s(s(0))))=
s(s(s(0+s^ (n,0)))) at tree position [n,1,1,1,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule2([1,1,1,2],1).

Current node is: [n,1,1,1,1,1]
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule2 applied 1 time(s): s^ (n,s(s(s(0))))=
s(s(s(s^ (n,0)))) at tree position [n,1,1,1,1,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: finished_tree.

General proof:

At position [] the sequent is s^ (n,0)+s(s(s(0)))=
s(s(s(0)))+s^ (n,0)
At position [n,1] the sequent is s^ (n,0+s(s(s(0))))=
s(s(s(0)))+s^ (n,0) using rule rule1([1],n)
At position [n,1,1,1,1] the sequent is s^ (n,0+s(s(s(0))))=
s(s(s(0+s^ (n,0)))) using rule rule1([2],3)
At position [n,1,1,1,1,1] the sequent is s^ (n,s(s(s(0))))=
s(s(s(0+s^ (n,0)))) using rule rule2([2,2,1],1)
At position [n,1,1,1,1,1,1] the sequent is s^ (n,s(s(s(0))))=
s(s(s(s^ (n,0)))) using rule rule2([1,1,1,2],1)

Type "yes" to see an individual proof, type "no" to
exit the system.
|: no.


yes
| ?- d.
==> /\x::x+s(s(s(0)))=s(s(s(0)))+x
by
omega_rule([rule1([1],n),rule1([2],3),rule2([2,2,1],1),rule2([1,1,1,2],1)])

yes
| ?- find_cut.

The expression, after applying rule1 n time(s), is:
s^ (n,_965+_959)=_485
The expression, after applying rule1 3 time(s), is:
s^ (n,_965+_959)=s(s(s(_2153+_2147)))
The expression, after applying rule2 1 time(s), is:
s^ (n,_959)=s(s(s(_2153+_2147)))
The expression, after applying rule2 1 time(s), is:
s^ (n,_959)=s(s(s(_2147)))
The generalisation is /\x::x+s(s(s(0)))=s(s(s(0)))+x % CUT FORMULA SUGGESTED
     % (IE. WE CAN DO INDUCTION
     % ON THIS ANYWAY!)
yes
```

Example Three: $\forall x\ x + s(x) = s(x) + x$

```
| ?- create_thm(1,user).
|: [] ==> /\x:: x+s(x)=s(x)+x.
```

```
yes
| ?- select(1).

yes
| ?- d.
==> /\x::x+s(x)=s(x)+x
by _

yes
| ?- omega_rule(_).
Which meta-variable do you wish to use?
If not known, type n.
|: n.
General assertion (to be proved) is s^ (n,0)+s(s^ (n,0))=s(s^ (n,0))+s^ (n,0).

Do you wish to input a full generalisation? (yes/no)
|: no.

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([2],1).

Current node is: []
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule1 applied 1 time(s): s^ (n,0)+s(s^ (n,0))=
s(s^ (n,0)+s^ (n,0)) at tree position [1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([1],n).

Current node is: [1]
Do you wish to select another position? (yes/no)
|: no.

Subexpression to be altered is: s^ (n,0)+s(s^ (n,0))
Do you wish to input subexpression after rule application? (yes/no)
|: no.

Subrule used: s^ (m,s^ (n-m,0)+s(s^ (n,0)))
New equation is, using rule1([1],n):s^ (n,0+s^ (n+1,0))=
s(s^ (n,0)+s^ (n,0))

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([1,2],n).

Current node is: [1,n,1]
Do you wish to select another position? (yes/no)
|: no.

Subexpression to be altered is: s^ (n,0)+s^ (n,0)
Do you wish to input subexpression after rule application? (yes/no)
|: no.

Subrule used: s^ (m,s^ (n-m,0)+s^ (n,0))
New equation is, using rule1([1,2],n):s^ (n,0+s^ (n+1,0))=
s(s^ (n,0+s^ (n,0)))

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule2([2,2,1],1).

Current node is: [1,n,1,n,1]
Do you wish to select another position? (yes/no)
```

```
|: no.

New equation is, using rule2 applied 1 time(s): s^ (n,s^ (n+1,0))=
s(s^ (n,0+s^ (n,0))) at tree position [1,n,1,n,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule2([2,2,1,2],1).

Current node is: [1,n,1,n,1,1]
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule2 applied 1 time(s): s^ (n,s^ (n+1,0))=
s(s^ (n,s^ (n,0))) at tree position [1,n,1,n,1,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: finished_tree.

General proof:

At position [] the sequent is s^ (n,0)+s(s^ (n,0))=
s(s^ (n,0))+s^ (n,0)
At position [1] the sequent is s^ (n,0)+s(s^ (n,0))=
s(s^ (n,0)+s^ (n,0)) using rule rule1([2],1)
At position [1,n,1] the sequent is s^ (n,0+s^ (n+1,0))=
s(s^ (n,0)+s^ (n,0)) using rule rule1([1],n)
At position [1,n,1,n,1] the sequent is s^ (n,0+s^ (n+1,0))=
s(s^ (n,0+s^ (n,0))) using rule rule1([1,2],n)
At position [1,n,1,n,1,1] the sequent is s^ (n,s^ (n+1,0))=
s(s^ (n,0+s^ (n,0))) using rule rule2([2,2,1],1)
At position [1,n,1,n,1,1,1] the sequent is s^ (n,s^ (n+1,0))=
s(s^ (n,s^ (n,0))) using rule rule2([2,2,1,2],1)

Type "yes" to see an individual proof, type "no" to
exit the system.
|: no.


yes
| ?- d.
==> /\x::x+s(x)=s(x)+x
by omega_rule([rule1([2],1),rule1([1],n),rule1([1,2],n),rule2([2,2,1],1),
rule2([2,2,1,2],1)])

yes
| ?- find_cut.

The expression, after applying rule1 1 time(s), is:
_87=s(_546+_544)
The expression, after applying rule1 n time(s), is:
s^ (n,_1092+_1086)=s(_546+_544)
The expression, after applying rule1 n time(s), is:
s^ (n,_1092+_1086)=s(s^ (n,_1638+_544))
The expression, after applying rule2 1 time(s), is:
s^ (n,_1086)=s(s^ (n,_1638+_544))
The expression, after applying rule2 1 time(s), is:
s^ (n,_1086)=s(s^ (n,_544))
The generalisation is /\x::x+s(v0)=s(x)+v0 % CUT FORMULA SUGGESTED

yes
```

Example Four: $\forall x\ x + s(x) = s(x + x)$

```
| ?- create_thm(p,user).
|: [] ==> /\x:: x+s(x)=s(x+x).

yes
| ?- select(p).

yes
| ?- d.
==> /\x::x+s(x)=s(x+x)
by _

yes
| ?- omega_rule(_).
Which meta-variable do you wish to use?
If not known, type n.
|: n.
General assertion (to be proved) is s^ (n,0)+s(s^ (n,0))=s(s^ (n,0)+s^ (n,0)).

Do you wish to input a full generalisation? (yes/no)
|: no.

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([1],n).

Current node is: []
Do you wish to select another position? (yes/no)
|: no.

Subexpression to be altered is: s^ (n,0)+s(s^ (n,0))
Do you wish to input subexpression after rule application? (yes/no)
|: no.

Subrule used: s^ (m,s^ (n-m,0)+s(s^ (n,0)))
New equation is, using rule1([1],n):s^ (n,0+s^ (n+1,0))=s(s^ (n,0)+s^ (n,0))

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule1([1,2],n).

Current node is: [n,1]
Do you wish to select another position? (yes/no)
|: no.

Subexpression to be altered is: s^ (n,0)+s^ (n,0)
Do you wish to input subexpression after rule application? (yes/no)
|: no.

Subrule used: s^ (m,s^ (n-m,0)+s^ (n,0))
New equation is, using rule1([1,2],n):s^ (n,0+s^ (n+1,0))=s(s^ (n,0+s^ (n,0)))

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule2([2,2,1],1).

Current node is: [n,1,n,1]
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule2 applied 1 time(s): s^ (n,s^ (n+1,0))=
s(s^ (n,0+s^ (n,0))) at tree position [n,1,n,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: rule2([2,2,1,2],1).
```

```
Current node is: [n,1,n,1,1]
Do you wish to select another position? (yes/no)
|: no.

New equation is, using rule2 applied 1 time(s): s^ (n,s^ (n+1,0))=
s(s^ (n,s^ (n,0))) at tree position [n,1,n,1,1,1]

Input "Rule(Pos of Subexpression, No of times applied)"
or "branch(Rule,Pos,No,[[S1],...,[SK]])" or "finished_tree" or "exit".
|: finished_tree.

General proof:

At position [] the sequent is s^ (n,0)+s(s^ (n,0))=
s(s^ (n,0)+s^ (n,0))
At position [n,1] the sequent is s^ (n,0+s^ (n+1,0))=
s(s^ (n,0)+s^ (n,0)) using rule rule1([1],n)
At position [n,1,n,1] the sequent is s^ (n,0+s^ (n+1,0))=
s(s^ (n,0+s^ (n,0))) using rule rule1([1,2],n)
At position [n,1,n,1,1] the sequent is s^ (n,s^ (n+1,0))=
s(s^ (n,0+s^ (n,0))) using rule rule2([2,2,1],1)
At position [n,1,n,1,1,1] the sequent is s^ (n,s^ (n+1,0))=
s(s^ (n,s^ (n,0))) using rule rule2([2,2,1,2],1)

Type "yes" to see an individual proof, type "no" to
exit the system.
|: no.


yes
| ?- d.
==> /\x::x+s(x)=s(x+x)
by omega_rule([rule1([1],n),rule1([1,2],n),rule2([2,2,1],1),rule2([2,2,1,2],1)])

yes
| ?- find_cut.

The expression, after applying rule1 n time(s), is:
s^ (n,_560+_554)=_88
The expression, after applying rule1 n time(s), is:
s^ (n,_560+_554)=fn9(s^ (n,_1445+_1439),_1809,_1810)
The expression, after applying rule2 1 time(s), is:
s^ (n,_554)=fn9(s^ (n,_1445+_1439),_1809,_1810)
The expression, after applying rule2 1 time(s), is:
s^ (n,_554)=fn9(s^ (n,_1439),_1809,_1810)
The generalisation is /\x::x+s(v0)=s(x+v0) % CUT FORMULA SUGGESTED

yes
| ?- halt.

Process prolog finished
```